Products for the
Global Information Centre

# SHARP APL/PC

I.P. Sharp
Associates

## Version 2
## February 1985
### SHARP APL Release 17

Microcomputer Products
I.P. Sharp Associates Limited
2 First Canadian Place
Suite 1900
Toronto, Ontario
Canada M5X 1E3

Telephone (416)364-5361
Telex 0622259 I P SHARP TOR
MAILBOX *pcq*

© 1985 by I.P. Sharp Associates Limited.

All rights reserved. This book or any portion thereof may not be reproduced without the written permission of I.P. Sharp Associates Limited.

Printed in Canada
February 1985
Publication Code: 0227 8502 E2
ISBN 0 86493 086 0

Contents

About This Book, 0-11
Conventions, 0-12

1 **Overview**, 1-1

Discusses SHARP APL/PC in perspective, and its software, the product itself, compatibility with other SHARP APLs, what you get, what you need to run it, and the services and support I.P. Sharp offers for it.

SHARP APL, 1-1
Parts of SHARP APL/PC, 1-2
Keyboard, 1-3
Compatibility, 1-3
What you get when you buy SHARP APL/PC, 1-4
What you need to run SHARP APL/PC, 1-4
I.P. Sharp services and support, 1-5

2 **Getting started**, 2-1

Discusses how to prepare your IBM PC for SHARP APL, and then how to start using it.

Copying the distribution diskettes to reference diskettes, 2-2
Making copies with two diskette drives, 2-2
Making copies with one diskette drive, 2-3
Making copies with a hard-disk drive, 2,4
Character sets and ROMS, 2-6,
Preparing the keyboard, 2-6
Reference diskette contents, 2-7
Building your APL diskette system, 2-8
Starting your APL system, 2-10
Starting your APL diskette system 2-10
Starting your SHARP APL hard-disk system, 2-10
Customizing your APL system, 2-12
BAT file rules, 2-12
Start parameters, 2-12
ACOM start parameters, 2-13
CP8X start parameters, 2-13
DOSX start parameters, 2-14
FSCR start parameters, 2-14
SFS start parameters, 2-15
SVP start parameters, 2-16

3 **Using SHARP APL/PC**, 3-1

Provides a joint introduction to three topics: using the main keyboard for entering characters that display on the screen; using the editing keypad for entry, revision, and display of new data, and display of data that has passed off the screen; and using APL itself, in terms of examples understandable to a beginner.

Experimentation, 3-3

Basic editing, 3-3
Sample APL session, 3-6
Further editing facilities, 3-12
Numeric keypad, 3-14
Alternate entries, 3-14
Miscellaneous, 3-15

**4 Using SHARP APL/PC files**, 4-1

Discusses how you can create SHARP APL files on your IBM PC, store data in such files, and retrieve data from them.

File identifiers, 4-2
Using files, 4-3
  Access control, 4-9
  Passnumbers, 4-11

**5 Communicating through the asynchronous ports**, 5-1

Introduces the Asynchronous Communications Auxiliary Processor by way of simple examples; and describes how to communicate with a SHARP APL/370 system.

Simple functions for using ACOM, 5-2
  Initializing the port, 5-5
  The Break command, 5-5
  Inquiring of and stopping the port, 5-5
  Sending and receiving characters, 5-5
  Connecting with the I.P. Sharp Network, 5-6
  Signing onto a SHARP APL/370 system, 5-7
  Signing off, 5-8
  Improving on these functions, 5-9
SHARP APL transfer software, 5-10
  Getting started, 5-10
  Switching between mainframe and PC, 5-11
  Transferring APL objects, 5-11
  Interrupting this session, 5-13
  Building other cooperating data transfer applications, 5-13

**6 Session Manager**, 6-1

Describes the concepts and facilities of the Session Manager. The Session Manager forms the interface between you and the SHARP APL/PC system.

Session log, 6-1
Screen areas, 6-2
Interpreting keyboard input, 6-2
Cases, 6-4
Keyboard input, 6-5
  Input stage, 6-5
  Typing characters, 6-5
  Overstriking, 6-7
  Caps reversal, 6-7
  Numeric lock, 6-7
  Editing input, 6-8

Moving the cursor, 6-8
Releasing the input stage, 6-9
Using the peg, 6-10
Moving the peg, 6-10
Grabbing a line, 6-10
Setting the reference line, 6-10
Scrolling the windows, 6-11
Selecting the current window, 6-11
Vertical scrolling, 6-11
Horizontal scrolling, 6-12
Status line, 6-13
Moving the status line, 6-14
Default state, 6-14
Changing the states, 6-14
Interrupting processing, 6-16
Using □*arbin* and □*arbout*, 6-17
The □*arbin* or □*arbout* argument, 6-17
The □*arbin* result, 6-18
Making keyboard input invisible, 6-19
Defining function keys, 6-20
Distributed workspace *fkeys*, 6-20
Using □*arbin* for function keys, 6-22

**7 Workspaces**, 7-1

Discusses SHARP APL/PC workspace identifiers, passwords, and sizes.

Identifiers, 7-1
Passwords, 7-1
Sizes, 7-2

**8 SHARP APL/PC File System reference**, 8-1

Describes the system functions you use to manage and access SHARP APL files on your IBM PC.

File identifiers, 8-1
SFS start parameters, 8-2
File system functions, 8-3
□*append*, 8-4
□*appendr*, 8-4
□*avail*, 8-4
□*create*, 8-4
□*drop*, 8-4
□*erase*, 8-4
□*fhold*, 8-5
□*hold*, 8-5
□*lib*, 8-5
□*names*, 8-5
□*nums*, 8-5
□*rdac*, 8-5
□*rdci*, 8-6
□*read*, 8-6
□*rename*, 8-6
□*replace*, 8-6

□*resize*, 8-6
□*size*, 8-7
□*stac*, 8-7
□*stie*, 8-7
□*tie*, 8-7
□*untie*, 8-7
File System events, 8-8
Physical storage considerations, 8-10
Summary of file functions, 8-12

**9 Shared variables and auxiliary processors**, 9-1

Discusses shared variables, the Shared Variable Processor, and how shared variables permit APL functions to make use of auxiliary processors.

Shared variables: a discipline of communication, 9-1
The Shared Variable Processor, 9-1
Sharing with non-APL programs, 9-2
Auxiliary processors, 9-2
Outline of how variable-sharing works, 9-3
"Sending" and "receiving", 9-4
Establishing a shared-variable link from the APL side, 9-4
Establishing and terminating a share, 9-5
Duration of sharing, 9-6
Clone ID: establishing your exact identity, 9-6
Inquiring about the degree of coupling, 9-7
Inquiring about offers from others, 9-8
Inquiring about the shared variable's state and controls, 9-8
Keeping track of who's set what, 9-9
Possible states of the shared variable, 9-10
Access control, 9-10
Effective access control, 9-12
Diagram summarizing shared variable communication, 9-15
Attempting to carry out an action that is inhibited, 9-16
Calculating in advance which actions will not be delayed, 9-17
Meaning of some possible settings of a variable's access control vector, 9-17
State-change signal, 9-17
Saving a workspace containing shared variables, 9-19

**10 User-written auxiliary processors**, 10-1

Discusses how you write and use user-written auxiliary processors.

The SVP, 10-1
The task dispatcher, 10-2
  Task posting, 10-2
SVP structures and terminology, 10-4
  PERPCV: per processor control vector, 10-4
  PERCMD: per SVP command, 10-5
  PERSCV: per share control vector, 10-5
  ASDATA: APL SVP data, 10-6
SVP operations, 10-7
SVP commands, parameters, and results, 10-8

Signing on: SVPON PERPCV, 10-8
Scanning for offers: SVPSVQ PERSCV, 10-9
Offering to share: SVPSVO PERSCV, 10-9
Retracting an offer: SVPSVR PO, 10-10
Access control, 10-10
Setting access control: SVPSVC PO,CV, 10-10
Reading the access state and access control: SVPSVS PO, 10-11
"Use" and "set", 10-11
Initial selection operations, 10-11
Data transfer operations, 10-11
Release control: SVPNDT PO, 10-12
Return codes, 10-13
Terminate but stay resident, 10-15
Task termination, 10-16
The MUSIC AP, 10-17
The MUSIC argument, 10-17
MUSIC initialization--MUSICEP, 10-17
MUSIC interrupt task--MYTICKINT, 10-18
MUSIC normal task--PLAY, 10-18
MUSIC termination--GOSHUT, 10-19

**11 DOS Interface Auxiliary Processor (DOSX), 11-1**

Describes DOSX. DOSX provides access to DOS files, parallel printers, some DOS function calls, memory (peek and poke), and user-written machine-language routines.

The shared variable, 11-1
The argument, 11-1
The result, 11-1
DOSX start parameters, 11-2
"Abort, Retry, Ignore?", 11-2
A simple session using DOSX, 11-3
DOSX commands, 11-4
DOS file commands, 11-5
Translation, 11-5
Current index, count, and translation, 11-6
Open file for read only, 11-6
Open file for read and write, 11-6
Create a file, 11-7
Read data from the file, 11-7
Write data to the file, 11-7
Size information, 11-8
Close the currently opened file, 11-8
Directory list, 11-8
Erase file, 11-9
Rename file, 11-9
Translate data, 11-9
Set translate table, 11-10
Printer support, 11-11
DOS function calls, 11-13
Select default drive, 11-13
Number of drives, 11-13
Version and DOSX information, 11-13
Make a subdirectory, 11-14

Remove a directory, 11-14
Change directory, 11-14
Memory commands, 11-15
Read from memory, 11-15
Write to memory, 11-15
Call user-written machine-language program, 11-16
The *call* argument, 11-16
The *call* result, 11-18
Writing and testing a machine-language program, 11-18
Using the assembler, 11-18
Using DEBUG and CALLTEST, 11-18
CTOH example, 11-19
CTOH.ASM listing, 11-20
Summary of commands, 11-22
Return codes, 11-23
Subcodes, 11-23

**12 Asynchronous Communications Auxiliary Processor (ACOM), 12-1**

Describes ACOM. An APL program uses ACOM to communicate with another system or device through the PC's asynchronous ports. ACOM can be used to communicate with a remote timesharing system, another PC, a printer, or a plotter.

Shared variables, 12-1
ACOM commands, 12-2
ACOM start parameters, 12-2
Start command, 12-3
Transmit/receive command, 12-6
Transmit Break command, 12-11
Stop command, 12-13
Inquire command, 12-14
Set/Read Frequency Table Command, 12-16
Return codes, 12-17
Modem Signals,12-20

**13 Full Screen Manager (FSCR), 13-1**

Describes FSCR, an auxiliary processor that handles the PC's monochrome or colour display separately from the Session Manager.

Compatibility with AP124, 13-1
Contrasting FSCR with the Session Manager, 13-1
Alternation between entry and display: full-screen dialogue, 13-2
A simple example, 13-3
Other experiments with the sample program, 13-4
Outline of how you pass instructions to FSCR, 13-5
FSCR's internal record of the screen and the screen you see, 13-6
Reading, writing, and fields, 13-6
The keyboard while you're using FSCR, 13-7
Typeahead during full-screen mode, 13-10
The variables that link your APL workspace to FSCR, 13-11

One-variable and two-variable interfaces, 13-11
How FSCR knows whether you're using one shared variable or two, 13-11
Choosing a one-variable or a two-variable interface, 13-12
Establishing contact, 13-12
Single-variable method, 13-13
Two-variable method, 13-14
The geography of fields, 13-16
Adjacent fields on the 3270, 13-17
Row-structure of a field, 13-17
Overwriting, insertion, and deletion in a field row, 13-18
Blanks and nulls at the end of a field row, 13-19
Effect of trailing nulls on insertion, 13-20
The blank-processing attribute of a field, 13-21
Cursor skip at the end of a field row, 13-21
Folded-vector fields, 13-22
The format matrix, 13-24
FSCR commands, 13-26
Cmd 1: Set field formats, 13-26
Cmd 1: Set new field formats, 13-26
Cmd 1: Reformat selected fields, 13-26
Cmd 9: Read the current format matrix, 13-26
Cmds 6, 7, 16, 24, and 26: Reset field attributes, 13-27
Cmds 2 and 4: Transmit data to be displayed, 13-27
Cmd 12: Set cursor position, 13-28
Cmd 11: Sound the alarm, 13-28
Cmds 35 and 37: Set character attributes, 13-29
Cmd 3: Write/wait/read, 13-29
Cmds 5, 30, 31, and 40: Reading what's on the screen, 13-30
Cmd 5: Read specified fields, 13-31
Cmds 30 and 31: Read modified fields, 13-31
Cmd 40: Read entire screen, 13-32
Checklist: points of difference between FSCR with SHARP APL/PC and AP124 with SHARP APL/370, 13-33
Distributed workspace *fscr*, 13-34
Command summary, 13-36
Return code summary, 13-37
Distributed workspace *fscrex*, 13-39
FSCR start parameters, 13-40

**Appendices**

A **Keyboard and character set**, A-1

Shows the actions of the main keyboard and control keypad, and how the SHARP APL character set is displayed.

Main keyboard, A-2
Control keypad, A-5
Control action by key, A-6
Character set, A-11

**B Customizing the keyboard and character set**, B-1

Describes how to customize your copy of the Session Manager for languages other than English, keyboard layouts other than North American, character ROMs other than I.P. Sharp's.

Session Manager tables, B-1
Hardware character table, B-1
Overstrike table, B-2
Keystroke table, B-2
Function key table, B-2
Workspace *sesm*, B-3
Sample session, B-4
Session Manager Actions, B-11
Summary of □arbin and □arbout codes, B-16

**C Union keyboard and Union Character Set**, C-1

Introduces the union keyboard; a combination of ordinary typewriter characters (ASCII terminal) and the characters used in APL programs.

**D Differences between SHARP APLs**, D-1

Discusses differences between SHARP APL/PC and SHARP APL/370.

You are the operator, D-1
You are the machine's only user, D-2
Only one APL task can run at a time, D-3
Workspace size, D-3
Session variables, D-4
Checklist of differences, D-5

**E Bibliography**, E-1

Lists some books you might find useful.

Expressly for the IBM PC, E-1
General texts on APL, E-1
Historical background, E-2
General topic treated in APL, E-2
Conference proceedings, E-4
Periodicals, E-4
Lightweight, E-4
About the PC, E-5

**F Utility software**, F-1

Lists and describes the utility software bundled with SHARP APL/PC. This appendix is intended for programmers of end-user products; it assumes familiarity and experience with APL, and SHARP APL in particular.

**G Installing the APL character ROM**, G-1

Describes how to install the APL character ROM in an IBM Personal Computer. This ROM defines APL characters for the video display; it replaces the character ROM that IBM supplies with the PC.

**Index**, X-1

## About This Book

This book is about version 2 of SHARP APL/PC, I.P. Sharp's implementation of the SHARP APL system for IBM personal computers.

SHARP APL/PC is supported for the IBM Personal Computer, Portable Personal Computer, Personal Computer XT, and Personal Computer AT. SHARP APL/PC is also supported for the Personal Computer XT/370 and AT/370, but does not take advantage of the System/370 personality.

This book is intended for any user of SHARP APL/PC: users of systems based on SHARP APL/PC, and designers and programmers of such systems; both experts in SHARP APL, and novices. No special knowledge of the IBM Personal Computer is assumed.

A bibliography (appendix E) is provided with references to other publications available. **Introduction to APL**, by K.E. Iverson, is recommended if you want to learn the APL language. This is complemented by the **SHARP APL Reference Manual**, which forms the definitive description of SHARP APL.

If you're more concerned with a specific application you don't need to know anything about APL, the first three chapters of the handbook are a useful supplement to your application's documentation.

SHARP APL/PC version 2 incorporates SHARP APL release 17.

## Conventions

In this book, keys are indicated by enclosing their markings in angle brackets. For example, the key marked with the letter S is indicated as "<S>".

Some keys are called by English words rather than by their symbols. Table 0-1 gives the key number, North American keytop marking, and handbook name. (Every key has a key number; see figure B-3 in appendix B.)

Table 0-1

**Key names**

| Key number | Keytop marking | Key name |
|---|---|---|
| 12 | - | minus |
| 13 | = | equals |
| 14 | ← | backspace |
| 15 | →\| | tab |
| 28 | [bent arrow] | return |
| 39 | ; | semicolon |
| 40 | ' | quote |
| 41 | ` | backquote |
| 42 | \ | backslash |
| 51 | , | comma |
| 52 | . | period |
| 53 | / | slash |
| 57 | [space bar] | space |
| **Control keypad** | | |
| 72 | ↑ | up |
| 74 | - | keypad-minus |
| 75 | ← | left |
| 76 | 5 | keypad-5 |
| 77 | → | right |
| 78 | + | keypad-plus |
| 80 | ↓ | down |

# Chapter 1: OVERVIEW

SHARP APL/PC is I. P. Sharp Associates' version of the SHARP APL system for IBM personal computers. SHARP APL/PC is supported for the IBM Personal Computer, Portable Personal Computer, Personal Computer XT, and Personal Computer AT. SHARP APL is also supported for the Personal Computer XT/370 and AT/370, but does not take advantage of the System/370 personality.

SHARP APL/PC version 2 incorporates SHARP APL release 17.

SHARP APL/PC is a full-scale language-processing system for IBM personal computers. You can use SHARP APL/PC to develop and execute application systems, and to operate your PC, its peripherals, and communications with other computers. SHARP APL/PC runs as a user program under the IBM Personal Computer Disk Operating System.

In particular, you can use the SHARP APL/PC system:

- to process numeric or character data with programs written in the SHARP APL programming language;
- to store and retrieve data, in a variety of formats;
- to operate peripheral devices, including a printer;
- to specify how keyboard actions are to be interpreted, and how the video monitor is to display information;
- to communicate with an "external device"--a serial printer or plotter, another PC, a remote timesharing system; or
- to pass data to, execute, and receive data from a non-APL program.

## SHARP APL

SHARP APL/PC interprets and executes statements written in the SHARP APL programming language. SHARP APL is a concise and powerful language; complicated operations can be performed in few statements, but a usable subset can be taught and used in less than a day (such a usable subset is taught in chapter 3). Not only can you store SHARP APL statements as programs for later or repeated use, but you can enter them in "calculator" mode and get immediate results.

SHARP APL on an IBM personal computer is exactly the same language as SHARP APL for System/370. Although some system functions, system variables, and system commands have been modified or redefined for the PC, all APL constructs available in one environment are available in the other: all primitive functions and operators; and all data types, including packages, complex numbers, and enclosed arrays. As well, two important segments of the system interface--event-trapping and files--remain the same. What this all means is that programs written in one environment can be moved to the other easily.

SHARP APL/PC is not affected by the PC's 16-bit architecture; for example, the maximum size of an array of data is limited solely by the memory available and not by the PC's internal restriction of 64KB. Further, there is no restriction on the size of an individual

axis of an array, other than that imposed by the workspace available.

## Parts of SHARP APL/PC

An operating SHARP APL/PC system comprises at least these program components:

- the IBM Personal Computer Disk Operating System;
- the SHARP APL interpreter;
- the SHARP APL/PC Session Manager;
- SHARP APL/PC system commands, system variables, and system functions;
- the SHARP APL/PC File System;
- the shared variable processor; and
- auxiliary processors, for: communicating with the PC's operating system; communicating asynchronously with external devices; and managing the full screen.

The **SHARP APL interpreter** is the essence of SHARP APL/PC. The interpreter is the language-processor: it accepts a statement in the SHARP APL language; interprets the statement symbol by symbol; executes each operation in the statement as it is come upon; and passes the result of the operation onto the next operation, or onto the user. Programs written in an interpreted language--such as SHARP APL--are far easier to debug than those in compiled languages; and the end user can readily--and powerfully--combine ad-hoc "calculator" operations with stored "program" operations.

The **Session Manager** controls input from the keyboard and output to the screen; records input and output transactions on its session log; and controls the log itself. You specify to the Session Manager how certain keybutton combinations are to be interpreted, and how certain characters are to be displayed. You can also program function keys. The Session Manager includes all usual input-editing facilities.

You can specify to the Session Manager what portions of the session log you want to see. You can grab recorded transactions from the log, edit them, and reenter them.

**System commands, variables, and functions** form the interface between SHARP APL and the PC. You use system variables or functions: to use packages (a SHARP APL data type); to trap or signal events (including errors); to control files; to control the current task; for special input/output; to report information about, and to control the active workspaces. As much as possible, the operations of the system interface are expressed in the terms of the APL language; thus, an APL statement can act on the system beyond SHARP APL.

SHARP APL in any environment gives you two ways to store data and programs: **saved workspaces** and **files**. A workspace is a named bundle of data and programs that can be loaded or copied, and instantly used. A file is a a collection of indexable, variable-length, self-described records, each of which is a SHARP APL data object.

The **shared variable processor** is a program outside APL that handles ad-hoc real-time communication between SHARP APL and an auxiliary processor through shared variables. A shared variable is a data object (of arbitrary type, shape, and size) that may be read and written by both a SHARP APL program and an auxiliary processor.

An **auxiliary processor** (AP) is a non-APL program that runs in the PC besides SHARP APL, and that SHARP APL controls and communicates with through the shared variable processor. You can write an AP to do anything the PC can do. The basic SHARP APL/PC product includes these APs:

- the DOS Interface AP;
- the Asynchronous Communications AP; and
- the Full Screen Manager.

The **DOS Interface Auxiliary Processor** (DOSX) gives an APL program access to the facilities of the Personal Computer Disk Operating System, for example: the use of DOS files; peeking and poking of memory; the use of DOS function calls (such as DIR); use of the parallel printer; and a generalized way to call non-APL programs.

The **Asynchronous Communications Auxiliary Processor** (ACOM) handles real-time communications with external devices through the PC's RS232 ports. Thus, a SHARP APL/PC program can communicate, through the worldwide standard asynchronous (start/stop) protocol, with, for example: another PC; a serially connected printer or plotter; a timesharing service, including a mainframe SHARP APL system and the I. P. Sharp Online Service. ACOM can handle two RS232 ports simultaneously.

### Keyboard

SHARP APL/PC introduces a new APL keyboard: the **union keyboard**. This keyboard is discussed in appendix C.

### Compatibility

SHARP APL/PC has been designed to be as closely compatible with SHARP APL/370 as possible. There are few differences, and these are minor and largely stem from the fact that the IBM PC is **not** an IBM System/370 mainframe. These differences do not impede the fast development or easy interchange of software between different SHARP APL environments.

For the end user, applications can appear and operate in exactly the same way as in SHARP APL/370.

For the APL programmer, the vast majority of the features available in SHARP APL/370 are found in SHARP APL/PC:

- The language is SHARP APL in full, incorporating all of I. P. Sharp's advances.

- Most of the system functions, system variables, and system commands are available; in particular: the ⎕*fmt* report formatter, ⎕*trap* and ⎕*signal* for event trapping, packages (⎕*pack* etc.), workspace storage and retrieval, and a component file system (⎕*read* etc.).

- A complete implementation of shared variables provides access to a selection of auxiliary processors which give the programmer a way to use the PC's peripherals.

The compatibility of SHARP APL systems and the notion of shared variables combine to support "hybrid" applications. Thus, some processing can be done by SHARP APL in a PC, and some by a SHARP APL system in a remote mainframe--with real-time communication between the two.

SHARP APL/PC version 2 is also completely compatible with SHARP APL/PCX version 1.

Appendix D **Differences between SHARP APLs** details differences among SHARP APL/PC, SHARP APL/PCX, and SHARP APL/370.

### What you get when you buy SHARP APL/PC

Each copy of SHARP APL/PC includes:

- two diskettes containing the SHARP APL/PC system, and utility programs;

- the **SHARP APL Pocket Reference**;

- this handbook;

- an APL character ROM; and

- APL stickers for the PC's keys.

As well, I.P. Sharp supplies a wide range of APL-related publications, including the **SHARP APL Reference Manual**, the complete description of SHARP APL.

Publications can be ordered separately; contact your I.P. Sharp representative, or the I.P. Sharp Microcomputer Products Department.

### What you need to run SHARP APL/PC

To use SHARP APL/PC, you need:

- An IBM Personal Computer, Portable Personal Computer, Personal Computer XT, or Personal Computer AT.

- At least 512K of main storage.

- A 5¼" double-sided diskette drive.
- The IBM monochrome display/parallel printer adapter; or the IBM colour/graphics adapter; and a monitor that functions with the adapter. Better results are achieved with the monochrome display/parallel printer adapter and the IBM monochrome display.
- An APL character ROM (installed on the display adapter). This ROM is not necessary if you already own the APL ROM sold by STSC Inc., or if you do not need APL characters.
- IBM Personal Computer Disk Operating System, version 2.0 or higher.

## I.P. Sharp services and support

As soon as you receive your copy of SHARP APL/PC, **immediately complete and return the registration card**. This registers you as a licensee of SHARP APL/PC and makes sure that you'll be informed of new products.

**Warranty**. SHARP APL/PC has a limited warranty against defects, as described in the warranty and license agreement shipped with SHARP APL/PC. Contact I.P. Sharp immediately if the product arrives damaged.

**Programming and consulting.** Every I.P. Sharp office throughout the world can supply you with programming, consulting, or education services on a contractual basis. I.P. Sharp offices are staffed by experts knowledgeable about APL and other I.P. Sharp products.

**Newsletter.** The I.P. Sharp Newsletter is issued six times a year, by free subscription. It contains topics of general interest to all I.P. Sharp customers, including SHARP APL/PC users. Completing and returning your registration card adds your name to the Newsletter mailing list.

**The online bulletin board.** If you are a customer of the I.P. Sharp Online Service, you can use the *pcnews* facility. *pcnews* contains topical items of interest to SHARP APL/PC users; anyone can submit suitable items to it. Contact any I.P. Sharp office if you'd like to join.

## Chapter 2: GETTING STARTED

This chapter discusses how to prepare your personal computer for SHARP APL, and then how to start using it.

Most of your preparations for using SHARP APL on your PC are the same as for any major software system: making copies of the distribution diskettes, and customizing your system so you can run APL day to day. Procedures for doing this are given in this chapter.

After you've read this chapter but before you do anything else, make the reference copies of the distribution diskettes. Otherwise, if your diskettes are damaged you risk losing your APL software with no way of recovering it.

Do not rush to start using the distribution diskettes directly to run your system.

You may also want to prepare your PC hardware for SHARP APL. You can install a ROM (an integrated circuit chip) to enable your monitor to display the SHARP APL character set. You can also install stickers on the fronts of many of the keys on the keyboard to indicate the location of special APL characters.

After you have
- made reference copies of the distribution diskettes,
- installed the character ROM (if you need or want to),
- installed the keyboard stickers (if you want to), and
- assembled and customized your APL system,

you will be ready to run SHARP APL on your PC.

## COPYING THE DISTRIBUTION DISKETTES TO REFERENCE DISKETTES

This section describes how to copy the SHARP APL distribution diskettes to reference diskettes.

The SHARP APL software is shipped to you on two double-sided 5¼" diskettes. These are the **distribution diskettes.**

Use the distribution diskettes only to make a set of **reference diskettes**: exact copies of the distribution diskettes. Once you've made the reference diskettes, store the distribution diskettes in a safe place. Use the distribution diskettes again only if your reference diskettes are damaged.

This section assumes some familiarity with the Personal Computer Disk Operating System and its commands. If you need more information than this section gives, see the IBM publications **Disk Operating System**, or **Disk Operating System User's Guide.**

You need two diskettes for making the copies. They need not be formatted. Any data on the diskettes will be lost, so be sure they do not contain anything important.

**Important:** Label these diskettes **"SHARP APL/PC Reference Diskette 1"** and **"SHARP APL/PC Reference Diskette 2"**, and transcribe the serial numbers from the labels on the backs of the distribution diskettes.

To make the copies, follow the directions in **Making copies with two diskette drives, Making copies with one diskette drive,** or **Making copies with a hard-disk drive.**

If you are quite familiar with DOS, or have special hardware facilities you may have better ways to make copies. But do make them, and do it carefully.

### Making copies with two diskette drives

This section assumes: that DOS is running; that drive A (the left-hand drive) is the default drive; and that the DOS FORMAT utility is available on your "DOS diskette". For more information about setting up your PC to meet these assumptions, consult your Personal Computer **Guide to Operations**, or your PC salesman.

1 Format the reference diskettes. Put your DOS diskette in drive A. Enter:

  A>format b:

  The FORMAT program is loaded; you're prompted to insert the diskette to be formatted.

2 Insert reference diskette 1 in drive B. Press <return>. When formatting is done, reply **y** to the prompt to format another diskette.

3 Insert reference diskette 2 in drive B. Press <return>. When formatting is done, reply **n** to the prompt to format another diskette.

4 Remove all diskettes.

5 Insert distribution diskette 1 in drive A and reference diskette 1 in drive B. Enter:

```
A>copy a:*.* b:
```

The names and number of files copied are displayed.

6 Remove both diskettes from the drives.

7 Insert distribution diskette 2 in drive A and reference diskette 2 in drive B. Enter:

```
A>copy a:*.* b:
```

The names and number of files copied are displayed.

8 Put write-protect labels on the reference diskettes. Store the distribution diskettes in a safe place.

### Making copies with one diskette drive

This section assumes: that DOS is running; that drive A (the left-hand drive) is the default drive; and that the DOS FORMAT utility is available on your "DOS diskette". For more information about setting up your PC to meet these assumptions, consult your Personal Computer **Guide to Operations**, or your PC salesman.

1 Format the reference diskettes. Put your DOS diskette in drive A. Enter:

```
A>format b:
```

The FORMAT program is loaded; you're prompted to insert the diskette to be formatted.

2 Remove the DOS diskette from drive A.

3 Insert reference diskette 1 in drive A. Press <return>. When formatting is done, remove the diskette from drive A, and reply **y** to the prompt to format another diskette.

4 Insert reference diskette 2 in drive B. Press <return>. When formatting is done, reply **n** to the prompt to format another format diskette.

5 Remove the diskette.

6 Insert distribution diskette 1 in drive A. Enter:

A>copy a:*.* b:

You are prompted to remove distribution diskette 1 (the **source diskette**) and insert reference diskette 1 (the **target diskette**). You will be prompted to exchange diskettes several times. Do so. When copying is done, the names and number of files copied are displayed.

7 Insert distribution diskette 2 in drive A. Enter:

A>copy a:*.* b:

You are prompted to remove distribution diskette 2 (the **source diskette**) and insert reference diskette 2 (the **target diskette**). You will be prompted to exchange diskettes several times. Do so. When copying is done, The names and number of files copied are displayed.

8 Put write-protect labels on the reference diskettes. Store the distribution diskettes in a safe place.

### Making copies with a hard-disk drive

This section assumes: that DOS is running; that drive C (the hard disk drive) is the default drive; and that the DOS FORMAT utility is available on drive C. For more information about setting up your PC to meet these assumptions, consult your Personal Computer **Guide to Operations**, or your PC salesman.

1 Format the reference diskettes. Enter:

C>format a:

You're prompted to insert the diskette to be formatted. (If FORMAT is not available on drive C, DOS displays "Bad command or file name". Load FORMAT from your DOS distribution diskette: see the **Guide to Operations**.)

2 Insert reference diskette 1 in drive A. Press <return>. When formatting is done, reply **y** to the prompt to format another diskette. Remove reference diskette 1 from drive A.

3 Insert reference diskette 2 in drive A. Press <return>. When formatting is done, reply **n** to the prompt to format another format diskette. Remove reference diskette 2 from drive B.

4 Insert distribution diskette 1 in drive A. Enter:

C>a:install

File INSTALL.BAT on the diskette is executed: subdirectory \SAPLPC is created and selected, and the files on the diskette copied into it.

5 You are prompted to remove distribution diskette 1 from drive A and insert distribution diskette 2. Do so, and press <return>. The files on this diskette are copied into \SAPLPC.

6 Remove the diskette from drive A. Execute the \SAPLPC\COPIER program. For example, if \SAPLPC is the current directory, enter:

```
C>copier
```

7 You're prompted to insert reference diskette 1. Do so, and press <return>. The contents of distribution diskette 1 are copied to the diskette.

8 You're prompted to insert reference diskette 2. Do so, and press <return>. The contents of distribution diskette 2 are copied to the reference diskette.

Put write-protect labels on the reference diskettes. Store the reference and distribution diskettes in a safe place.

## CHARACTER SETS AND ROM'S

The characters that appear on an IBM PC's monitor are defined by a read-only memory (ROM) chip on the adapter board that runs the monitor. To have the SHARP APL character set, you replace this chip with one you get with SHARP APL/PC. How you replace the chip is described in appendix G.

The APL character ROM is similar to the character ROM supplied by IBM. Both ROMs have the same letters, numerals, and punctuation marks. However, the APL ROM replaces special non-APL characters with special APL characters. For detailed information about the characters defined by each ROM, see appendix B **Customizing the keyboard and character set**.

If you use the IBM-supplied character ROM, the special APL characters do not display in their proper form. Although a given APL character does not display the proper graphic, it does display a unique one. Careful use of table A-1 (in appendix A) will help you to decipher APL expressions without the APL character ROM. If you use the IBM-supplied character ROM you can use most end-user software packages from I. P. Sharp without difficulty: most application software restricts its use of APL-unique characters; as a matter of policy, **all** application software written by I. P. Sharp is intended to be usable on both APL and non-APL devices. But if you want to display or write APL programs, you will find the lack of APL characters a serious impediment to your work.

If you use the APL character ROM, you can display or write APL expressions clearly and easily. However, you might find that some non-APL programs on your IBM PC display inappropriate (but unique) characters.

The I. P. Sharp APL character ROM chip is similar to the APL character ROM sold by STSC Inc. If you've already installed STSC's ROM, you don't need to install I. P. Sharp's.

If you've installed someone else's APL character ROM, you can use the facilities described in appendix B to use its characters.

### Preparing the keyboard

To indicate which keys elicit which special APL characters, you can mark the keys with stickers. These stickers are provided with SHARP APL. To apply them, just follow the instructions printed with them.

## REFERENCE DISKETTE CONTENTS

Your SHARP APL reference diskettes include these files:

```
ACOM.EXE
APL.BAT
COPIER.BAT
CP8X.EXE
DOSX.EXE
FSCR.EXE
INSTALL.BAT
SAPL
SFS.EXE
SVP.EXE
```

ACOM.EXE
is the Asynchronous Communications Auxiliary Processor (ACOM).

APL.BAT
is a batch file of DOS commands to start a complete APL system.

COPIER.BAT
is a file of DOS commands to create reference diskettes from subdirectory \SAPLPC.

CP8X.EXE
is the SHARP APL supervisor; it contains the System/370 emulator, the Session Manager, and the APL supervisor. (In SHARP APL/PC version 1, this file was called SUPER.EXE.)

DOSX.EXE
is the DOS Interface Auxiliary Processor (DOSX).

FSCR.EXE
is the Full Screen Manager (FSCR).

INSTALL.BAT
is a file of DOS commands to install SHARP APL on a hard disk, and to create reference diskettes.

SAPL
is the SHARP APL language processor (the same code as on any SHARP APL/370 system, including the I.P. Sharp Online Service).

SFS.EXE
is the SHARP APL File System.

SVP.EXE
is the Shared Variable Processor.

Your reference diskettes also contain APL workspaces, APL files, and other files. (For a complete list, use DOS's DIR command.) APL workspaces are identified by the file name extension .SAW, and APL files by the file name extension .SAF.

## BUILDING YOUR APL DISKETTE SYSTEM

This section discusses how you can build SHARP APL/PC system diskettes. If you use a hard disk, you will already have built your system when you executed the INSTALL program.

If your APL system is to run from diskettes, then you may want to have the basic DOS system files on the first diskette. They don't take up much room on the diskette and they make it possible for you to load DOS from APL system diskette 1.

This section assumes that you have two diskette drives. If you have only one drive, modify the procedure accordingly.

You need to format two diskettes for use as APL system diskettes. Format APL system diskette 1 with the /S option so that it contains the DOS system files. Format both system diskettes with the /V option, and give them appropriate volume labels. Also, **put human-readable labels on the diskettes when they are formatted.**

1 Insert a DOS system diskette in drive A (which is assumed to be the default drive).

2 To format APL system diskette 1, enter:

```
A>format b: /s/v
```

3 Insert APL system diskette 1 in drive B.

When you're prompted for the volume label, answer with something appropriate, such as **APLSYSTEM1**.

When formatting is done, reply **n** to the prompt for another format.

If you intend to load the DOS system from APL system diskette 1, you may want to copy your standard AUTOEXEC.BAT file and any special initialization programs (for example, a program to set the clock automatically) to that diskette now.

4 Remove APL system diskette 1.

5 To format APL system diskette 2, enter:

```
A>format b: /v
```

6 Insert APL system diskette 2 in drive B.

When you're prompted for the volume label, answer with something appropriate, such as **APLSYSTEM2**.

When formatting is done, you're prompted to format another diskette. You should probably also format a third diskette. If so, reply **y** to FORMAT's prompt. This diskette will contain your APL workspaces, APL files, and other data. Give this diskette an appropriate volume label (for example, **APLDATA**).

7 APL system diskette 1 is to contain:

```
ACOM.EXE
APL.BAT
CP8X.EXE
DOSX.EXE
FSCR.EXE
SFS.EXE
SVP.EXE
```

These files can be copied from reference diskette 1 to APL system diskette 1 by copying APL.BAT and all the .EXE files. To do this, insert reference diskette 1 in drive A, APL system diskette 1 in drive B, and enter:

```
A>copy a:apl.bat b: /v

A>copy a:*.exe b: /v
```

The specified files are copied to APL system diskette 1.

8 Remove the diskettes.

9 APL system diskette 2 is to contain only file SAPL. Insert reference diskette 2 in drive A, APL system diskette 2 in drive B, and enter:

```
A>copy a:saplpc b: /v
```

10 You will probably want to customize and modify your APL system diskettes; at least for now do not cover their write-notches to protect their contents. This does mean you have to be careful.

You only need to use the APL system diskettes to start the APL system. After it is started, you can remove the system diskettes, so you can use the drives for diskettes with APL workspaces, and APL and non-APL files.

11 Now copy the APL workspaces and files to the APLDATA diskette from the reference diskettes. Insert reference diskette 1 in drive A, the APLDATA diskette in drive B, and enter:

```
A>copy a:*.sa? b: /v
```

Because the file name extensions for SHARP APL workspaces and files are SAW and SAF, the DOS file specification *.SA? includes all APL workspaces and files.

12 Insert reference diskette 2 in drive A, and enter the above command again.

## STARTING YOUR APL SYSTEM

### Starting your APL diskette system

1 Load your DOS system in the usual way; this could be from APL system diskette 1.

2 Put APL system diskette 1 in drive A, and system diskette 2 in drive B. If you have only one drive, wait until DOS prompts you for drive B.

3 Enter:

```
A>apl pb:
```

This executes all of the statements in the APL.BAT file: it starts SVP, DOSX, ACOM, FSCR, SFS, and CP8X. Each prints a brief message. The parameter "PB:" indicates that the SAPL file is to be loaded from drive B (see the discussion of start parameters, below).

Loading APL takes less than a minute. When it is finished, the system identification is on the screen and your system is ready for input.

4 Remove both APL system diskettes, and insert the APLDATA diskette in drive A. This gives you access to the distributed workspaces, and space for your own workspaces and files.

5 When you terminate APL (for example, with *)off*), you should probably still have the APLDATA diskette in drive A so that files will be properly closed. DOS then asks you to insert a system diskette in drive A. Insert your APL system diskette 1 in drive A, so that DOS can reload the part of itself that was overlaid by APL and can complete execution of the APL.BAT file.

6 Move to chapter 3 to learn about the system. You can then return to this chapter when you are ready to customize your system.

### Starting your SHARP APL hard-disk system

1 Load your DOS system in the normal way. It is assumed that the default drive is the hard disk and that it contains the SAPLPC directory. If these assumptions are not correct, then you must make the appropriate changes to the following steps.

2 Execute the batch file APL.BAT in directory \SAPLPC:

```
C>chdir \saplpc
C>apl capldata
```

(If subdirectory \SAPLPC had already been chosen, you can omit [theits] CHDIR command.) This starts APL with the

\APLDATA subdirectory as the default location for workspace and file operations.

3 The loading process takes about thirty seconds and reports each step it takes. When it is finished, the system identification is on the screen and your system is ready for input.

## CUSTOMIZING YOUR APL SYSTEM

There are many ways to customize your system, and many tools to do it with. As you learn more about your hardware, DOS, and APL, you will find ways to work more effectively and productively.

BAT files are a powerful tool in tailoring your APL system to your needs:

- You can have parameters to the BAT file allowing you to specify options at start time.
- You can create different BAT files for different purposes.
- You can use DOS commands and other programs before and after the APL commands. They may establish the environment for APL, and then establish a different environment when APL ends.
- BAT files can invoke BAT files. For example, on a hard-disk system, you could include the two statements for starting APL into an APL.BAT file in the root directory (which is a different file from the APL.BAT file in the \SAPLPC subdirectory).
- You can decide which programs (auxiliary processors) to start.
- You can decide what parameters to supply to each of the programs that make up the system.

Study the section on batch files in IBM's Disk Operating System manual in order to be able to take advantage of BAT parameters and other facilities.

### BAT file rules

There are some simple rules about what programs constitute a valid APL system and in what order they must be started:

- The SVP **must** be started first if **any** auxiliary processor or SFS is to be run.
- The order of the auxiliary processors and SFS is not important.
- CP8X must be started last.

If you don't intend to use a program, then don't start it. For example, if your application doesn't require FSCR, don't start it, so the memory otherwise used by FSCR will be available in your APL workspace.

### Start parameters

A start parameter is indicated by a single letter that is immediately followed by its value (if any). For example:

```
SFS T6 B10000
```

This starts the File System with 6 as the maximum number of files you can tie, and 10000 bytes for buffers.

### ACOM start parameters

**Ln**
specifies that asynchronous port **n** is attached to an I.P. Sharp linedriver. **n** may be 0 or 1 or blank (meaning both ports). If you omit L, you indicate that neither port is attached to a linedriver.

**Qn**
indicates whether ACOM is to display its start message. **n** may be 0 (specifying that the message is not to be displayed) or 1. If you omit Q, the message is displayed.

**Un**
specifies ACOM's processor number. **n** may be any value from 2 to 999. If you omit U, ACOM's processor number is 700.

### CP8X start parameters

**Bn**
specifies the size of the session log, in bytes. **n** may be any integer from 0 to 64000. If you omit B, the session log is 4096 bytes long.

**C**
specifies a subdirectory to be chosen. Specify this parameter to make sure that the current directory is the directory containing your APL workspaces, APL files, and other data. If you omit C, no subdirectory is chosen.

**H**
indicates that the **first** load of a workspace does **not** check the password. Specify this parameter to regain access to a workspace you were locked out of. Only the first load is affected. You can also specify the the W parameter to name the workspace you wanted loaded with the help of the H parameter. This facility is analogous to *loadhelper* in SHARP APL/370. If you omit H, workspace loads are subject to normal access control.

**In**
specifies the maximum length of the input line, in bytes. If you omit I, the input line may be 512 bytes long.

**Px**
specifies or modifies the path to the file containing the APL language processor program. The file specification coded in CP8X is "SAPL"; so, for example, if PB: is specified, the file is B:SAPL. If you omit P, the path is not modified.

**Un**
specifies the user number for the APL task. **n** may be any integer from 1000 to 2147483647. If you omit U, the user number is 3000; this default value has no inherent meaning.

**Wx**
specifies a workspace to be automatically loaded when APL is started. Embedded blanks are not allowed in parameters, so if a drive is specified it is right next to the name (for example, W3PROFILE or W2MYWS:LOCK). If you omit W, the APL session starts in a clear workspace.

### DOSX start parameters

**Bn**
specifies the length of the largest character vector that can be set in a variable shared with DOSX. **n** may be any integer from 200 to 32767. If you omit B, the largest settable character vector may be 2000 characters long.

**Qn**
indicates whether DOSX is to display its start message. **n** may be either 0 (which specifies that the message is not to be displayed) or 1. If you omit Q, the message is displayed.

**Un**
specifies the processor number to be used by DOSX. **n** may be any integer from 2 to 999. If you omit U, DOSX uses processor number 710.

**Sn**
specifies the number of variables that can be shared at once with DOSX. **n** may be any integer from 1 to 255. If you omit S, up to 4 variables can be shared at once with DOSX.

**Wn**
specifies the printer width (see **Printer support** in chapter 11 for more information about this parameter). **n** may be any integer from 1 to 32767. If you omit W, the printer width is 80.

### FSCR start parameters

**Bn**
specifies the size of the buffer used by FSCR, in bytes. This buffer is used for all the storage requirements of FSCR, including the screen image and the shared variables. **n** may be any integer from 1000 to 63530. If you omit B, the buffer has 15000 bytes.

**Fn**
specifies the method used by FSCR in writing to a colour/graphics screen. **n** may be 0, 1, or 2. If you omit F, method 2 is used.

**F0**
indicates the fastest method, but results in disconcerting screen flashes on most monitors.

**F1**
indicates a fancier but slower method that has less noticeable flashes than F0.

**F2**
indicates a method that writes to the screen only during the horizontal flyback and has no flashes, but is slower than F0 or F1. With F2, a fast typist can get ahead of the display.

In most cases, you will want F2. However, if your colour monitor is sophisticated enough to avoid the flashes itself, then you would want to use F0 or F1.

**Mn**
indicates whether FSCR is to use monochrome attributes, even though the device appears to DOS as a colour monitor. **n** may be either 0 (specifying that FSCR is not to use monochrome attributes) or 1. If you omit M, FSCR does not use monochrome attributes for what appears to be a colour monitor. The M parameter may be useful with some special, non-IBM display adapter boards.

**Qn**
indicates whether FSCR is to display its start message. **n** may be either 0 (specifying that the message is not to be displayed) or 1. If you omit Q, the message is displayed.

**Un**
specifies the processor number to be used by FSCR. **n** may be any integer from 2 to 999. If you omit U, FSCR uses processor number 124.

### SFS start parameters

**Bn**
specifies the size of the buffer used by the File System, in bytes. **n** may be any value from 2500 to 32767. This buffer is used for all the storage requirements of the File System.

**Hn**
indicates whether the first file tie is to be helped. **n** may be either 0 (specifying that file ties are not be helped) or 1. If you omit H, file ties are not helped.

H1 indicates that the **first** file tie with `⎕tie` does **not** check the access matrix, and ties the file with full access. Use this to regain access to a file that you are locked out of. When you tie a file this way, you should immediately correct the access matrix. Only the first tie is affected, so you may need to change the P parameter to CP8X to avoid having any ties done before you have a chance to tie the desired file. This facility is analogous to *`filehelper`* in SHARP APL/370.

Qn
indicates whether the File System start message is to be displayed. **n** may be either 0 (specifying that the message is not to be displayed) or 1. If you omit Q, the start message is displayed.

Tn
specifies the number of files you can have concurrently tied. **n** may be any integer from 1 to 100. If you omit T, you can have 4 files concurrently tied.

**SVP start parameters**

Bn
specifies the total memory space available for shared variables whose value is unreferenced by a partner in a share. The largest variable that can be shared is this value less some overhead. The space indicated by this parameter can be used by one large variable, or by several smaller ones. **n** may be any integer from 100 to 65530. If you omit B, the shared-variable buffer has 3000 bytes.

Pn
specifies the maximum number of concurrent SVP processors. APL and the APs are processors; SFS is not. **n** may be any integer from 2 to 200. If you omit P, 4 SVP processors may run concurrently; this allows APL, DOSX, ACOM, and FSCR.

Qn
indicates whether SVP is to display its start message. **n** may be either 0 (specifying that the message is not to be displayed) or 1. If you omit Q, the message is displayed.

Sn
specifies the number of variables that can be shared at once. **n** may be any integer from 2 to 200. If you omit S, 6 variables can be shared at once.

## Chapter 3: USING SHARP APL/PC

This chapter introduces three topics:

- Using the main keyboard (extending from the <Esc> key at the upper left, to <CapsLock> on the lower right; illustrated in figure A-1 in appendix A).
- Using the editing keypad.
- Using APL itself.

For more information about these topics, consult these sources:

- For a summary of all keyboard actions, see appendix A **Keyboard and character set**.
- For detailed technical information about how the keyboard and display are handled, see chapter 6 **Session Manager**.
- For more information about the APL language, see: the **SHARP APL Pocket Reference**; **Introduction to APL**; or the **SHARP APL Reference Manual**.

Read this chapter at the computer: learn by doing.

Any characters you enter appear on the screen and cause no other action until the entire entry is **released** to the computer by pressing the key marked with an arrow with a bent shaft (called <return> in this handbook). Verify this by pressing the top-row keys <9> and <minus> and <8> in that sequence, then **release** the entry and note that the correct result of subtraction is displayed on the screen.

The effect of each key on the **main keyboard** is to enter one of the characters marked on the key: the lower character in **base case** (lower case); and the upper character in **shift case** (upper case), that is, when holding down either of the two <shift> keys marked by large vertical arrows.

- Enter:

```
2+3+4
+/ 2 3 4
9-(5-2)
```

The <**backspace**> key (marked ← and located just above <return>) steps the cursor back and erases the characters. Experiment with its use for correcting an entry in preparation.

You can obtain a number of less familiar symbols in an **alternate case**, by holding down the <Alt> key to the left of the space bar. (These symbols may be marked by stick-on labels on the fronts of the keys.) The entire alternate case of characters is:

```
¨ ¯ < ≤ = ≥ > ≠ ∨ ∧ × ÷
? ⍵ ∈ ⍴ ~ ↑ ↓ ⍳ ○ * ← →
⍺ ⌈ ⌊ _ ∇ ∆ ∘ ' ⎕ ⊢ ⊣ ⋄
⍎ ⊂ ⊃ ∩ ∪ ⊥ ⊤ | ⍌ ⍋ ⍒
```

- Use base and alternate cases to enter:

```
8×4
8÷4
⍳5
×/⍳5
```

You can use the key marked <CapsLock>, instead of the <Alt> key. Use <CapsLock> to enter:

```
a←5
2×a
⍳a

a
b←⍳4
b

+/b
×/b
p← 2 3 5 7

p×b
p+b
sides← 4 5

area←×/sides
perimeter←2×+/sides
```

You can display the value assigned to a name (such as *sides* above) by entering the name. Verify this for the names used above. Also enter the expressions:

```
AREA←3
2+area-AREA
```

to verify that capital letters may be used in names.

The <backspace> key in alternate case steps the cursor back but does not erase, leaving the character to be overstruck by the character next entered. If this overstriking produces one of the valid composite APL characters, it remains on the screen. Otherwise the attempt to overstrike has no effect.

- By entering o followed by <Alt backspace>, followed by |, form and release the following entry:

```
⌽ 1 2 3 4 5
```

For a picture of the main keyboard in base, alternate, and shift cases, see figure A-1 in appendix A; or the **SHARP APL Pocket Reference**.

### Experimentation

From this point on you may wish to experiment more freely by entering anything that occurs to you, particularly if it promises to clarify the examples given. However, if you go too far afield, you may waste time puzzling over matters that will come more easily if you proceed through the text. Moreover, you may get the keyboard into a state that is very confusing because it concerns matters not yet covered.

It is therefore important to learn first how to reset any possible state and return the keyboard to its **virgin state** (that is, its state when you first start APL), using the following steps:

- Press <Esc>. This restores all **editing states** to their virgin values.
- Twice press the key labelled <Break> (on its front) while holding down the key marked <Ctrl>. This interrupts any computation or display in progress.
- Enter *)clear*. This step erases all names that have been assigned, so you should only use it as a last resort.

You're certain to encounter error messages when you experiment. Most are self-explanatory, as in *domain error* for 4÷0, and *value error* for 3+*abcd* when *abcd* has not been assigned a value. For a description of all APL error messages, see the **SHARP APL Reference Manual**.

### Basic editing

The area at the bottom of the screen that is released to the computer by the <return> key (marked by the bent arrow) is the input stage. Editing actions provided by the keyboard make it easy to change what's there, and to copy lines from the log.

The most commonly used editing actions are:

- Moving the **cursor**, which determines the area of the stage to be affected by subsequent keystrokes;
- Moving the **peg**, which marks a line in the log for possible copying to the stage;
- **Scrolling** the **window** to display any portion of the log, for perusal or for marking by the peg.

Editing facilities are controlled by the **editing keypad**, located to the right of the <return> key. Any discussion of editing refers to this portion of the keyboard. The APL editor attaches special meanings to the editing keys; don't confuse them with main keys bearing similar symbols.

The cursor control keys--the keys in the editing keypad marked with arrows--are called

<up>
<left> <right>
<down>

according to the direction in which the arrow points.

- Enter 2+4+5. Before you release it, use the <left> and <right> keys to move the cursor over the entry. Then position the cursor on the 4, press the <Ins> key, and enter 3+ (on the main keyboard) to obtain the entry 2+3+4+5. Then release the entry.

- Repeat the foregoing, noting that the characters 3+ are no longer inserted before the 4, but rather replace the characters 4+.

- Begin an entry, move the cursor back a few steps, then depress each of the keys <Del> and <backspace>, and observe their different modes of deletion.

The behaviour of the keyboard is determined by the **states** of certain toggles or counters. These states are displayed in the **status line** at the top of the screen. You can learn the meanings of the status line just by observing the effects of your normal keyboard use. Begin with the following experiments:

- Depress <Ins> repeatedly, and note that a caret (the symbol ∧ ) appears and disappears in one of the status line's fields. Verify that the appearance of the caret indicates that the **insert** (rather than **replace**) mode is in effect in the stage.

- Press <Ins> to make the caret appear. Prepare an entry, and release it. Observe that the caret disappears.

- Depress <Alt> and <Ctrl> simultaneously and note that CAPS appears in the status line.

- Enter an expression that includes letters, and note that they appear as capitals without your having to press a <shift> key. In fact, the way <shift> affects letters is now reversed: you press <shift> to get lowercase letters.

- Depress <Alt> and <Ctrl> together again to restore normal behaviour.

- Change one or more states from their virgin condition, and verify that the <Esc> key restores them.

The peg is initially obscured because it occupies the same row of the screen as the status line. The peg can be moved up or down by the keys <PgDn> and <PgUp>.

- First press <Esc> to restore the virgin state and enter *)clear*.
- Enter (and release) the expression 2+3+4+7+11.
- Move the peg down over the expression just entered, and press the <keypad-minus> key (on the right) to copy it down to the stage.
- Move the cursor, replace the 4 by 5, press <Ins>, and insert *a*← at the beginning. Then release the entry.
- Enter ⍳25, place the peg over the **result** produced, and copy it to the stage.
- Edit the stage to insert *b*← at the beginning, to delete the numbers 1 and 3, and to append the ten numbers 100 to 109 to the expression; then release the entry.
- Repeat the two preceding steps, but use the <down> key to move the cursor down to the second row of the stage for entering the last ten numbers. Use the <up> key to move the cursor back to the first row of the stage and verify that the cursor will move it no farther. Also verify that the <down> key will continue to move the cursor down, entering rows of blanks in the stage.

In alternate case, the <up> key scrolls the window (rather than moving the cursor) so as to display portions of the log that have already passed off the screen.

- Use <Alt up> to scroll the window up several steps, and use <Alt down> once or twice to verify that it scrolls it down.
- Use the peg to grab a line from the log that had passed off the screen.
- Release the line and note that the window returns to normal state, displaying the newly produced entry and result.

(You can get rid of the peg by pushing it off the screen with <PgUp> or <PgDn>.)

There are a number of further elementary editing conveniences that may be worth experimenting with at this stage:

- Most keys repeat if held down. This includes editing keys such as <backspace> and the cursor control keys.
- The <tab> key, just below the <Esc> key, moves the cursor to the right in eight-space steps; in shift case, it moves it to the left.
- The <Home> key moves the cursor to the beginning of the stage, and the <End> key moves it to the end; that is, to the farthest position occupied by the cursor during the current entry.
- In alternate case, the <Del> key deletes all the way to the end position; in shift case, it deletes all the way to home.

Before proceeding to the more advanced editing facilities treated in a later section, it may be well to exercise those already presented. The next section presents a sequence of APL expressions which can serve for exercise in the use of the keyboard as well as provide an informal introduction to APL for anyone not familiar with it, and to novel features of this implementation of APL for anyone who is.

## Sample APL session

Press <Esc> to restore the keyboard to its virgin state; enter *)clear* to destroy all existing names.

Enter the indented expressions listed below; the unindented results are shown in the first few examples only, since you can observe them on the screen. The note on the right states the general point that the example is intended to illustrate, and provides page references [in square brackets] to the **SHARP APL Pocket Reference**. Consult the **Pocket Reference** and experiment with further expressions to clarify or generalize the point at issue.

Be sure to display the value of each name assigned (by entering it alone) even though display is not explicitly included in the example. Also, use every opportunity to exercise the editing facilities, as, for example, in recalling and editing an earlier similar expression rather than making a complete new entry. For example, if you release an expression that contains a mistake, you don't have to type in the expression again; instead, you can move the peg to the mistaken expression, press the keypad-minus key to copy it to the input stage, revise it in the stage, and release it.

If you have to suspend your work in the session, start again here.

```
      area←4×5                    Names may include either case [3]
      pH←¯5                       Negative sign for negative constants
                                  [3]
      pH-3                        Subtraction sign [5]
¯8
      144e¯2                      Exponential form of constant [3]
1.44E¯
      p← 1 2 3 4                  Four-element vector (list) [3]
      p×p
1 4 9 16

      p×pH                        Scalar applies to all elements
¯5 ¯10 ¯15 ¯20
      ⌽p                          Reversal [6]
4 3 2 1
      q←'lever'                   Character constant [3]
      ⌽q                          Reversal
```

Many of the expressions which follow differ little from one to the next, and therefore provide opportunities to exercise the peg to copy a line down to the stage for revision and release.

```
j←⍳5                    Index generator [6]
j∘.×j                   Function tables:   Times [14]
j∘.-j                                      Subtraction

j∘.÷j                                      Division
j∘.⌈j                                      Maximum
pow←j∘.*j                                  Power

+/j                     Summation [12]
+\j                     Cumulative sums
×/j                     Product

×\j                     Progressive products
+/pow                   Row sums
+⌿pow                   Column sums

pow+.×j                 Matrix product [14]
j+.×pow
pow+.×pow

5⍴1                     Five ones
+\5⍴1                   Integers to 5
+/+\5⍴1                 Sum of integers to 5
+/+\10000⍴1             Takes about 20 seconds
```

You don't have to wait for the last entry to be processed before you release another entry. In fact, you can release several entries before the last entry is processed. Verify this by using the peg to reenter the last line above several times rather quickly.

When you do this, the word IN in the status line (which indicates that input will be accepted) is replaced by R (for Run) when work is in progress, and that the R may be followed by a digit that indicates the number of entries awaiting execution.

A simple example of **function definition** follows. The function defined here is used later in the discussion of text processing.

```
      5 1 ⍴ j                  Produce a one-column table [7]
      ((⍴j),1)⍴j

      ∇ z←COL y                A name can be given to any
[1]   z←((⍴y),1)⍴y             expression or sequence of
[2]   ∇                        expressions [45]

      COL j                    Use the defined function COL.
```

Some simple examples of **text processing** follow: breaking a text into words, finding their average length, and sorting them.

```
      text←⍞                    Character input [29]
There lived a man as I've been told
      text←text,' ',⍞
in the wonder working days of old

      ' ',text                  Display with a space prefix
      words←¯1⊂' ',text         Cut into words [15]
      ⍴words                    Number of words [6]
      ⌽words
      ⍴⍤1⊃words                 Lengths of individual words
      (+⌿⍴⍤1⊃words)÷⍴words       Average length
      table←↑COL words          Format 1-column table [8]
      ⍴table
15 7

      alph←' ',⍞                Prompt for character input; type it
                                in:
abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
      ⍴alph
53
      sorted←table[alph⍋table;]     Sorted table [7,9]; ⍋
                                    formed by overstriking
                                    Δ |
```

The following examples use a distributed SHARP APL file called *nations*. The examples assume that the file *nations* is on the current directory of the default drive. If you followed the startup procedures in chapter 2, this will be true.

```
      ⎕lib 0                    Display the names of the APL files on
                                the default disk.

      'nations' ⎕tie 3          Assign the number 3 to the file
                                nations.

      ⎕read 3 1                 Read component 1
      ⎕read 3 2                 Read component 2
      titles←⎕read 3 1
      data←(⎕read 3 2),(⎕read 3 3)

      titles,data

      pop←⍤1⊃data[3;]           Indexing [7]
      grid←(⌈/pop)×.025×⍳40
      pop∘.≥grid

      '.⎕'[1+pop∘.≥grid]        Indexing by a table
      ctry←data[1;]
      ctry                      List of countries
      ⍴ctry
```

```
⍕COL ctry              Table of countries [7,8]
(⍕COL ctry),'.□'[1+pop∘.≥grid]   Barchart
(⍕COL ctry),'.□'[1+pop∘.≥⌽grid]  Reverse barchart
```

The <keypad-minus> key copies the pegged line down to the stage after clearing the stage; the <keypad-plus> key does not clear the stage, but copies the line down to the position of the cursor. It can therefore be used to compose an entry from two or more pegged lines. The next example provides an opportunity to exercise this facility.

```
13⍴ 1 0
(13⍴ 1 0)⍀(⍕COL ctry),'.□'[1+pop∘.≥⌽grid]
                                   Double-spaced
⍉(13⍴ 1 0)⍀(⍕COL ctry),'.□'[1+pop∘.≥⌽grid]
                                   Vertical

abtitles← 4 3 3 4 4 3 ↑¨>titles  Abbreviate titles
abtitles □replace 3 1            to replace title.
□read 3 1              Read back to verify.

□untie 3               Close file.

area←>data[4;]
density←pop÷area
density>40

(density>40)/ctry      List of dense countries [13]
⍋density               Grade [8]
ctry[⍋density]         Ordered by density
```

DOSX is one of several **auxiliary processors** with which you may **share** a variable name; you assign values to this shared variable, which the auxiliary processor then accepts as arguments, and assigns its result to the name; that is, you and the auxiliary processor take turns assigning a value to the shared variable. In the case of DOSX, the result is the indicated portion of a DOS file, and it therefore provides access to the native disk operating system.

```
□untie □nums           To avoid interaction with the APL
                       file system, untie all APL files.

710 □svo 'y'           Share the name y with the file
                       processor [23].

y←'dir *.*'            Ask for DOS directory.
y                      Display directory. (The first six
                       characters are a status code. DOSX
                       always includes a status code when it
                       sends you information.)
```

```
      y
<Ctrl Break Break>
sv interrupt
      y
      ^                          If you try to read the shared
                                 variable again--without setting it for
                                 a new command--you'll wait
                                 indefinitely. You can break this wait
                                 by pressing the <Break> key twice
                                 while holding down <Ctrl>.

      y←'openro nations-.saf'
                                 Open for read only--suffix saf
                                 indicates that this is a SHARP APL
                                 file.
      y
00 00;                           Code indicates success.
      y←'read 0,1500,0'          Beginning at 0, read 1500 bytes,
                                 with no translation.

      y
00 00;⎕nations etc.              Only the APL names in this are
                                 readable, the rest requires a
                                 character translation.
      y←'close'                  Close the file nations.saf
      y
00 00;

      y←'create dosfile'         Create a DOS file called dosfile.

      y
00 00;                           Return value of zerosindicates
                                 success

      y←'write,0;Now is the time ' Write without translation

      y←'write,1;Now is the time ' Write with translation

      y←'read 0,16,0'            Read first 16 without translation
00 00;Now is the time
      y←read 16,16,2'            Read next 16 with translation
      y
00 00;Now is the time
      y←'read 0,32,0'            Read entire file
      y
      y←'read 0,32,2'
      y
      y←'close'

      y←'erase dosfile'

      y←'openrw prn:'

      cr←⎕av[⎕io+156]

      y←'write,5;Print the APL expression *o 0j1',cr
```

(Prints only if printer is attached; code 5 provides translation appropriate to the printer; if there's no printer, the computer will wait for about less than a minute before returning an error code.)

```
y←'close'
⎕svr 'y'                    Withdraw share.
```

It is often necessary to **format** a table so that a form that is more readable, with numbers rounded to a specified number of decimal points, with specified numbers of spaces allotted to each column, with specified indicators added to negative numbers, and with specified decorators provided between columns. The format function ⍕ does this:

```
t←(⍳6),¯500+.001×? 6 4 ⍴1e6
t
2⍕t                         Two decimal places for all columns
                            [8]

15 2 ⍕t                     15 spaces per column
a← 2 0 9 2 6 0 9 3 7 1
a⍕t                         Space and precision of each column

e←'5* 55,555.50 55000.55 00000.00'
b←(⍳5), 5 3 ⍴ 23456.123 78.4
b

e⍕b                         Format by example [10]
e                           Compare the format control with the
                            formatted table above it.
```

**System commands** (which all begin with a right parenthesis) provide information about, and control of, the environment in which APL expressions are executed. A complete list is given in the **Pocket Reference**. These are the more common cases:

```
)vars                       Gives a list of variable names
)erase a t
)vars                       a and t have been erased

)save abcd                  Save this workspace under the name
                            abcd on the default drive
)clear
)vars                       All names have been erased

)load abcd                  Load workspace abcd to restore
                            earlier status
)vars
)fns                        List defined functions (should
                            include COL defined earlier)
```

## Further editing facilities

This section presents further facilities for controlling the window introduced in the discussion of basic editing, and shows how to open and use a second, static window at the top of the screen.

The <Home> and <End> keys that move the cursor to the beginning and end of the **stage** may be used in alternate case to move the window to the beginning and end of the **log**.

- Press the <Esc> key to restore the keyboard to virgin state.
- Press <Alt Home> to observe the movement of the window. Note that the home position may not show the beginning of the session; space allocated to the log is limited, and the earliest lines are dropped as needed.
- Press <Alt End> and then move the peg down over the screen, observing how the **log line number** in the status line indicates the line of the log on which the peg rests.
- Press <Alt Home> and set the peg just below the status line; the value of the log line number is now the number of lines already dropped from the log.
- Press <Alt down> to scroll the window down over the log.
- Again press <Alt Home> and review the entire log, but this time by using the fast page-at-a-time scroll provided by <shift down> and <shift up>.

This window is **dynamic** in the sense that release of a new entry restores it to its normal position at the end of the log. Show this by first pressing <Alt Home> to move the window away from the end of the log, and then releasing the entry `⍳9`. Note that when you press <Alt Home>, "SC" appears in the status line.

Although this dynamic behaviour is often convenient, it is not so when you want to use the peg and the <keypad-minus> key (as described in **basic editing**) to grab a **sequence** of lines from some portion of the log. Convenient means for this are provided in two ways: you can set a **reference line** that is moved to by pressing <shift Home>, and set a **scroll lock** to keep the window at a fixed portion of the log.

- Scroll the window to some mid position between home and end, and set the peg over an easily recognized line.
- Press <shift keypad-5> (on the editing keyboard) to establish the pegged line as the reference line.
- Press either <Alt Home> or <Alt End>, and then press <shift Home> to move to the established reference line.
- Use the reference line and peg movement to grab a sequence of lines from the reference-line portion of the log.

Scrolling of the window may be **locked** so that the release of further entries does not cause the window to move to display the new entry and its result. Consequently, successive entries simply disappear from the screen, although they are appended to the log and will be displayed after the scroll is unlocked.

- Watch the status line for the appearance of LK when <ScrollLock> is pressed.
- Enter and release ⍳9 and note the behaviour.
- Press <ScrollLock> again and note how the end of the log is maintained exactly as it would have been in the absence of locking.

**The upper window.** The status line is moved up and down by <Alt PgUp> and <Alt PgDn>. The screen area above the status line is the upper window. This window is static, and the scroll lock therefore has no effect on it; however, all other controls apply equally to the upper and the lower windows. The window affected at any time is the one to which the vertical arrow in the status line points; its initial state is down, and the state is reversed by <Alt keypad-5>.

- Use <Alt PgDn> and <PgUp> to move the status line to about midscreen.
- Press <Alt keypad-5> to point the window indicator upward, and then repeat on the upper window the various experiments already performed on the lower window.
- Verify that both the status line and the peg have offscreen positions.

**Wrapped and extended display.** Long displays must normally be wrapped to fit the width of the screen, but a window that can scroll to right and left gives complete views of unwrapped or **extended** displays.

- Press <Esc> and enter *)clear* to ensure that the screen is returned to initial state.
- Prepare and release an entry that occupies more than one line of the stage, beginning with *b*← and continuing with a long sentence enclosed in quotes.
- Observe the folding of the line just released, then press <Alt NumLock>, and observe that the folding has disappeared, and that only the first part (eighty columns) of the line is shown.
- Press <Alt right> to scroll the window to the right to display the rest of the extended line. Note the significance of the changing counter in the status line and of the * (that indicates the existence of some unseen portion still further to the right).
- Press <Alt left> and <shift right> and <shift left> to move the window further to right and left.

- Press <Alt NumLock> to reestablish wrapping, and to move the peg onto the long line.

The last experiment above illustrates the fact that the peg marks an **entire line**; any use of the <keypad-minus> key to grab a line marked by the peg also takes the entire line.

- Press <Esc> to assure that folding has been restored; then enter and release *b* to display the value assigned to *b* by the entry in the preceding sequence.

- Press <Alt NumLock> and note that although the line that produced *b* extends, the display of *b* does not.

- Verify that the display of *b* comprises separate lines by restoring folding and moving the peg down over the display.

The reason that the display of *b* consists of separate 80-character lines is that the output of *b* was controlled by a print width of 80 as determined by the initial default value of ⎕*pw*. By setting ⎕*pw* to its maximum value of 250, we can observe the extended display of (new) output as well as input lines.

- Press <Esc> and enter ⎕*pw*←250.

- Enter 3 50 ⍴⍳150 and observe the display.

- Move the peg down over the rows of the three-row table produced.

- Press <Alt NumLock>, and use scrolling to view the entire matrix in its extended display.

### Numeric pad

Pressing the <NumLock> key converts the editing keypad into a **numeric keypad** convenient for entering numeric data, including data that contains negative signs and periods. Since the keys in this mode do not provide the editing facilities, it might be well to experiment with this mode so as to learn to recognize the symptoms, even though you do not plan to use it. A second depression of the <NumLock> key restores normal behaviour.

### Alternate entries

A few symbols can be entered in more than one way, and one way may sometimes be more convenient.

The alternate case provides (in a single keystroke) four characters that might otherwise be entered as composites requiring three keystrokes each:

## Miscellaneous

When work is in progress, the word IN in the status line is replaced by R (for Run), and the R may be followed by a digit that indicates the number of entries stacked awaiting execution. Stacked entries may be deleted by entering upper <Del>. Entering <Ctrl Break> (also labelled <ScrollLock>) interrupts execution, immediately if entered twice, and otherwise at the end of the current line.

Escape from character input of the form $a\leftarrow$⍞ (often denoted by $o\text{-}u\text{-}t$) is provided by <shift ScrollLock>. <Alt ScrollLock> sets a **curb**, causing the word to show in the status line, and stopping the next attempted output when it occurs. The fact that output is currently being withheld by a curb is indicated by C in the run (R) position. A curb is reset by the same keys.

## Chapter 4: USING SHARP APL/PC FILES

This chapter discusses how you can create SHARP APL files, store data in such files, and retrieve data from them.

A SHARP APL file is a collection of indexable, variable-length, self-described records, each of which is a SHARP APL data object. You store or retrieve data and programs from SHARP APL files with executable SHARP APL statements. Each of these records is a **component**.

A file is **similar** to a workspace in several respects:

- It has a name.
- It resides in your Personal Computer's **mass storage**, that is, on a diskette or hard disk.
- It may be created, used, and modified over time, and erased when it is no longer needed.
- A component of a file, like a variable in a workspace, may be any APL value--any scalar, vector, matrix, higher-dimensional array, package, or general array--holding either characters (including the character representation of a function) or numbers. The only limit on the size of a component or variable is the implicit one that it should fit in a workspace and on a disk.

A file is also **different** from a workspace in some important respects:

- More than one file may be active at the same time.
- A file can hold large or small amounts of data. The reservation of a file may be specified by the owner and can range from a few thousand bytes up past the capacity of the drive. The size of a workspace is not directly variable.
- A file is manipulated by system functions which can either be typed at the keyboard or executed within APL programs.
- A file owner may specify access control for a file.
- A variable in a workspace is referred to by its name. A component in a file is referred to by its **component number**, which gives the component's position within the file. Component numbers are consecutive positive integers, ranging from the number of the first component up to the number of the last component.

**File functions** bring bring components into the active workspace for processing, and save values from the active workspace as components of a file. File functions are performed with SHARP APL system functions; for example, *□read* reads a component from a file, and *□append* appends to a file a value from the active workspace.

Improper use of file functions leads to **file errors**. Such errors are indicated in the usual way by error reports. Chapter 8 **SHARP**

**APL/PC File System reference** describes each file function and the file errors it may produce.

The syntax of the file functions shown in this chapter omits the optional passnumber, since it is used only in more complex applications. Chapter 8 gives the full syntax of the functions, including the use of passnumbers.

## File identifiers

A **file identifier** comprises a drive number and a file name.

The **drive number** is a nonnegative integer indicating the disk drive where a file was, is, or will be. If it is omitted or 0, the default DOS drive is implied. File operations are always performed in the **current directory** of the specified or default drive.

Drive numbers correspond to DOS drive identifiers as follows:

| Drive number | DOS drive identifier |
|---|---|
| 0 | default |
| 1 | A |
| 2 | B |
| 3 | C |
| 4 | D |
| ... | ... |

The **file name** comprises 1 to 8 letters or numerals, beginning with a letter; only the first alphabet--*a b c*, etc.--may be used. If the drive number is present, it is separated from the file name by one or more blanks. Leading and trailing blanks are permitted.

These are valid file identifiers:

```
'      1 aardvark   '
'a'
'3 a'
'alpha123'
'x9y8z7'
```

Every SHARP APL/PC file is a DOS file. The DOS filename is formed by padding the file name with hyphens to eight characters (if necessary) and adding .SAF to it. Thus, what SHARP APL sees as file *zap*, DOS sees as file ZAP-----.SAF. (A SHARP APL file's DOS file name is padded with hyphens to avoid any conflict with DOS reserved names, such as PRN.)

## USING FILES

Before you work with a file, you must make an association between the file identifier and a positive integer **file number**. Operations on the file refer to the file by its number, rather than by its name. The particular value of the file number is unimportant as long as it is different from any other file numbers being used at the same time. The association between a file identifier and a file number is called a **file tie**, and a file together with its associated file number is said to be **tied**.

Descriptions and examples of file functions in this chapter use the two files below. If you enter the definitions of the programs used here, you can execute each example in turn.

The file named *persons* has four components, each a vector of characters. The file *sales* also has four components, each a vector of numbers.

| component number | *persons* | *sales* |
|---|---|---|
| 1 | *'smith'* | 5 6 3 1 4 |
| 2 | *'jones'* | 2 6 10 |
| 3 | *'kelley'* | 4 6 2 9 1 5 |
| 4 | *'becker'* | 20 6 4 |

A new file is brought into existence with the □*create* function, and new components are placed on the file with the □*append* function. The following program builds *persons* and *sales*. (The files are built in the current directory of the default drive. If this is a diskette drive, you must insert a formatted diskette in the drive.)

```
    ∇ prog1;pmat;i;name
[1]   'persons' □create 5
[2]   'sales' □create 20
[3]   pmat←4 6⍴'smith jones kelleybecker'
[4]   i←1
[5]  loop:name←pmat[i;]
[6]   'enter values for ',name
[7]   ⍞ □append 20
[8]   ((name≠' ')/name) □append 5
[9]   i←i+1
[10]  →(i≤1↑⍴pmat)/loop
[11]  □untie 20 5
    ∇
```

The □*create* function establishes a new file, and makes it ready for further functions. The syntax is:

```
'fileid' □create filenumber
```

The file identifier must be different from that of any existing file, and the file number must not be in use. The file is created in the current directory of the specified or default drive. The reservation of a newly created file is 102,400 bytes.

When *prog*1 is executed, line 1 creates the file name *persons* and ties it to file number 5. The file *persons* now exists, but is empty: it contains no components yet. Components are added to the file *persons* by the ⎕*append* function on line 8.

The ⎕*append* function puts a new component, containing an APL value from the active workspace, at the end of a file. The value in the workspace is not altered. The syntax is:

value ⎕*append* filenumber

The left argument is the value of the new component, and may be the name of a variable or the result of any other APL expression. The right argument is the file number of the file being appended to (and may itself be the result of an APL expression).

Examples of appending to files *sales* and *persons* appear in lines 7 and 8 of*prog*1.

The ⎕*untie* function has the syntax:

⎕*untie* filenumbers

⎕*untie* breaks the association between a file and its file number. The argument is a vector of zero or more file numbers, so ⎕*untie* can untie several files at once, as in line 11 of *prog*1. ⎕*untie* has no effect on the values stored in a file.

WARNING

**Untie all files on a diskette before you remove it from the drive.** The PC's memory contains critical information about tied files. If you remove a diskette that holds a tied file, the file may be damaged unpredictably. If you mistakenly remove a diskette and then reference it, DOS prompts you with an "Abort, Retry, Ignore" message to give you a chance to recover; you can reinsert the diskette, and reply Retry. If you have already inserted another diskette, or if you insert the wrong diskette after the "Abort" message, the original file on the first diskette may be damaged, and the new diskette may also be damaged. The system command )*off* unties all files.

An existing file that is not tied can be made available for processing, as shown in the next program. This program prints each of the components of the file named *persons*, along with the sum of the numbers in the corresponding component of *sales*. The two file functions introduced in this example are ⎕*tie* and ⎕*read*.

```
    ∇ prog2;x;i
[1]   'persons' ⎕tie 1
[2]   'sales' ⎕tie 2
[3]   'name       sum'
[4]   i←1
[5]  loop:x←⎕read 2,i
[6]   (10↑⎕read 1,i),⍕+/x
[7]   i←i+1
[8]   →(i≤4)/loop
    ∇
```

The ⎕*tie* function is used to establish an association between a file identifier and a file number. In *prog2*, *persons* is tied to file number 1, and *sales* is tied to file number 2. The syntax of ⎕*tie* is:

```
'fileid' ⎕tie filenumber
```

In *prog1*, *persons* is tied to file number 5; in *prog2*, *persons* is tied to file number 1. File numbers may be chosen at will, and the number for a particular file can be different in different ties. The file you specify must be in the **current directory** of the specified or default drive.

The ⎕*read* function copies the value of a file component into the active workspace. The value may be placed in a variable, as in line 5 of *prog2*; it may be used in an expression, as in line 6; or it may be displayed directly. The syntax of ⎕*read* is:

```
result←⎕read filenumber componentnumber
```

The argument of ⎕*read* is a two-element vector: the first element is the file number, and the second element is the component number. "**result←**" in the syntax shows that ⎕*read* yields an explicit result and hence may appear within a larger expression. This is not true of functions, such as ⎕*tie*, that have no explicit result.

Program *prog2* ties the files *persons* and *sales*, but does not untie them, so those two files remain tied as files 1 and 2. A file tie is maintained until you untie or erase the file, or until you sign off.

The status of tied files is not changed by any system commands except )*off* and )*continue*. Loading a saved workspace, or clearing the active workspace leaves active any ties established earlier in this session. You can load different workspaces to process a file without having to re-establish the tie.

You list the names of the SHARP APL files on a particular drive with the ⎕*lib* function. The syntax is:

```
result←⎕lib drivenumber
```

The result is a character matrix. Each row holds the identifier of a file in the designated drive. The expression:

```
⎕lib 1
```

produces this result:

```
1 persons
1 sales
```

Each row of the matrix is a file identifier. The number of APL files in drive A is equal to `1↑⍴⎕lib 1`.

When a file is no longer needed, it can be erased:

```
'fileid' ⎕erase filenumber
```

The file name is deleted, all components are destroyed, the space they occupied is made available for use by other files, and the file tie is broken. If you want to erase a file it must be tied and you must specify both the file name and the file number.

The following program rearranges filed information. The components of *persons* and *sales* are merged into a new file, *records*. Each odd-numbered component of *records* comes from *sales*. The two original files are erased after the function is finished. (*persons* and *sales* are still tied following execution of *prog2*.)

```
    ∇ prog3;i
[1]   'records' ⎕create 3
[2]   i←1
[3]  loop:(⎕read(1+2|i-1),⌈i÷2) ⎕append 3
[4]   →(8≥i←i+1)/loop
[5]   ⎕untie 3
[6]   'sales' ⎕erase 2
[7]   'persons' ⎕erase 1
    ∇
```

Line 3 shows how an APL expression can be used to calculate the value of a file number or a component number. In this example, the file numbers 1 and 2 did not appear in a *⎕untie* because erasing *persons* and *sales* also unties them.

Two niladic functions, *⎕names* and *⎕nums*, report on the status of file ties. *⎕names* has the syntax:

```
result←⎕names
```

and returns a matrix of characters, in which each row is the drive number and name of a file currently tied. (it has the same format as *⎕lib*'s return.)

*⎕nums* returns a vector of the file tie numbers corresponding to the names given by *⎕names*. The syntax of *⎕nums* is:

```
result←⎕nums
```

Both *⎕names* and *⎕nums* identify files in the order in which they were tied.

The simplest way to untie all tied files is by executing the statement *⎕untie ⎕nums*, since the vector returned by *⎕nums* is exactly what *⎕untie* needs in order to untie all files.

The function *⎕appendr* puts a new component, containing a value from the active workspace, at the end of the file and returns the component number of the component. The syntax is:

```
result←value ⎕appendr filenumber
```

A file component can be replaced by a new APL value, using the *⎕replace* function. The syntax is:

```
value ⎕replace filenumber componentnumber
```

For example, to replace the second component of file number 1 by the reciprocals of the first fifty positive integers, this statement could be used:

```
(÷⍳50) ⎕replace 1 2
```

The new value of a component may be any APL value that fits in the active workspace and fits in the available space on the disk. Any additional file storage that is needed to hold the new value is provided automatically.

A common use of *⎕replace* is to update a component by modifying its value. In this example, the value of the variable *debit* is catenated to the existing value of component 3 of file number 7.

```
((⎕read 7 3),debit) ⎕replace 7 3
```

File size can increase disproportionately if many small components are replaced by larger components. For a discussion of file storage, see chapter 8 **SHARP APL/PC File System reference**.

The *⎕drop* function removes components from either end of a file. The *⎕drop* function does **not** free any space on the drive. File space is freed only when the file is erased or copied.

The syntax is:

```
⎕drop filenumber n
```

If **n** is positive, **n** components are removed from the beginning of the file; if **n** is negative, **|n** components are removed from the end of the file. If **n** is 0, no components are removed from the file. The component numbers of the remaining components are not changed; for example, if file 99 has ten components numbered 1 through 10, then after executing:

```
⎕drop 99 4
```

the file has six components remaining, numbered 5, 6, 7, 8, 9, and 10. A further execution of:

```
      □drop 99 ¯2
```

leaves components 5, 6, 7, and 8.

Components can not be dropped from the interior of a file. Other techniques may be used to signify that an interior component holds no information and should be bypassed in later processing; the simplest is to replace the component by an empty vector, ⍬ or ''.

The function □*size*, with syntax:

```
result←□size filenumber
```

returns a four-element vector. The first element is the component number of the first component of the file. The second element is the component number of the last component plus 1. The third and fourth elements are the amount of file storage currently occupied (the **file size**) and the file reservation, in bytes. A new file has a reservation of 102,400 bytes. File size and file reservation are always set in units of 512 bytes. A newly created file uses 1,024 bytes.

The □*rename* function is used to change a file identifier. The syntax is:

```
'fileid' □rename filenumber
```

For example:

```
'reports' □tie 100
'oldrep' □rename 100
□untie 100
```

Note that □*rename* does not create a second copy of a file--after the execution of the previous example, *reports* is no longer on the disk, but *oldrep* is.

You cannot use □*rename* to copy a file from one drive to another.

The function □*resize* is used to change the reservation of the file. The syntax is:

```
reservation □resize filenumber
```

For example, this changes the reservation of *picayune* downward to about ten thousand bytes:

```
'picayune' □tie 30
1e4 □resize 30
```

The resulting reservation is 10,240 bytes. The actual reservation is rounded upward to the next multiple of 512 bytes.

Along with the value stored in a component by appending or replacing, three other pieces of information are carried with each component of a file: the work area neeced to hold the component's

value; the user number of the user that last replaced or appended the component; and the time that the component was last replaced or appended. The *⎕rdci* function reads this **control information**. The syntax is:

```
result←⎕rdci filenumber componentnumber
```

The result is a three-element vector holding the described information. The time, in the third element of the result, represents sixtieths of seconds since 00:00 1 March 1960.

You can use functions in the distributed *utility* workspace to decode the timestamp. For example:

```
      'zap' ⎕tie 101
      ⎕←¯1↑⎕rdci 101 1
4.652903738e10
      )load util
saved  1984-04-08 12:34:56
      ftt ¯1↑⎕rdci 101 1
1984/09/26 12:12:36.405
```

Three functions--*⎕stie*, *⎕hold*, and *⎕fhold*--are supported for compatibility with the SHARP APL/370 File System, where several tasks can access the same file concurrently. For SHARP APL/PC, *⎕stie* (share-tie) is identical to *⎕tie*, and *⎕fhold* and *⎕hold* have no effect.

The function *⎕avail* indicates whether the File System is available. The syntax is:

```
result←⎕avail ⍳0
```

*⎕avail* requires a dummy right argument of ⍳0 or '' to allow for future extension.

If the File System is available, *⎕avail* returns 1; if not, it returns 0. If the File System isn't available, this probably means you didn't start it when you started APL; for more information, see chapter 2 **Getting started**.

## Access control

Access control can prevent inadvertent misuse of a file, but it offers little protection against an expert and malicious user.

Associated with every file is an **access matrix** that records which user numbers are authorized to use the file, and which file functions each user number is allowed to access it by. An access matrix is integer-valued, with three columns and no more than nine rows. Each row represents authorization for a single user number. The first element of the row is the user number; the second is an encoding of permitted file functions; and the third is a passnumber.

The user number may be that of any individual, or 0--which represents all user numbers.

When a file is tied, the access matrix for the file is searched for a row describing the user's user number and passnumber. If a row is found whose first **and** third elements match the user number and passnumber respectively, or--failing that--if a row is found whose first element is zero **and** whose third element matches the passnumber, then the authorized forms of access are taken from column 2 of that row. Any attempt to use a file function that isn't authorized signals a *file access error*. If the value for *⎕tie* is not included in column 2, the *⎕tie* itself will fail with a *file access error* report.

The owner of a file has complete access to the file with a passnumber of 0 unless some row of the access matrix contains the owner's user number. In such a case, the owner's access is limited by the file functions designated. No one but the owner has access to a file unless his user number matches the user number or a 0 in column 1 of the file's access matrix; if a user tries to tie a file to which he has no access, *file access error* is signalled.

If you lock yourself out of a file, you can use a File System start parameter to regain access. For information about this, see the description of SFS start parameter H in chapter 2 **Getting started**.

Functions like *⎕names* and *⎕nums* are not restricted by access matrices and never signal *file access error*, because they are not functions that are performed **on** a particular file. Also, there is no restriction on *⎕untie* or *⎕stie*; these functions are permitted if **any** other function is authorized.

The functions *⎕stac* and *⎕rdac* manipulate the access matrix. The value of the access matrix is set by the *⎕stac* function. Its syntax is:

value *⎕stac* filenumber

The left argument must be an integer-valued three-column matrix with nine or fewer rows. It replaces the previous value of the access matrix. Initially, when a file is created, its access matrix has no rows--its shape is 0 3. The file owner has complete access with passnumber 0.

The access matrix is read into the active workspace by the *⎕rdac* function. The syntax is:

result←*⎕rdac* filenumber

The result is the access matrix for the file designated by the right argument.

The second element in a row of the access matrix is the sum of coded values for the particular functions that are allowed. The values are:

| | |
|---|---|
| ¯1 | all |
| 1 | □*read*, □*size*, □*stie* and □*untie* |
| 2 | □*tie* |
| 4 | □*erase* |
| 8 | □*append* |
| 16 | □*replace* |
| 32 | □*drop* |
| 64 | □*hold* |
| 128 | □*rename* |
| 256 | □*rdac* and □*stac* |
| 512 | □*rdci* |
| 1024 | □*resize* |
| 2048 | □*fhold* |
| 4096 | □*rdac* only |
| 8192 | □*stac* only |
| 16384 | □*appendr* |
| 32768 | □*size* |

Thus, a value of 10 indicates authorization to execute □*tie* and □*append* functions on this file. A value of ¯1 indicates authorization for all functions.

Negative numbers deny certain accesses (that is, /"permit everything but"). In general, ¯1-n (or -1+n) will restrict accesses specified by n. For example, ¯5 permits all but □*erase* (although it does permit □*stac*, so the file can be erased with a two-step procedure).

The third element in a row of the access matrix, the **passnumber**, is usually 0. Nonzero passnumbers are used only to exercise detailed control over file access, and are discussed next.

### Passnumbers

Detailed control over file access may be achieved by requiring a nonzero passnumber. A user must tie the file using the passnumber that appears with his user number in the access matrix. Furthermore, most other file functions acting on the file must provide that passnumber. Chapter 8 **SHARP APL/PC File System reference** describes which functions require passnumbers.

Omitting a passnumber from an argument (as in all uses of file functions earlier in this chapter) is equivalent to providing an explicit passnumber of 0. A mismatching passnumber signals *file access error*.

Passnumbers are intended for use within locked APL functions that the application designer makes available to his users.

A simple use of passnumbers is to protect a file from accidental modification. For example:

```
      'zap' □create 101
      □←am← 2 3 ⍴ 3000 1 0 3000 ¯1 13
3000    1    0
```

```
3000   ¯1   13
      am ⎕stac 101
```

Tying *zap* with no passnumber allows only read and size operations. In order to modify or erase the file, you have to tie it with passnumber 13.

A more complicated use of passnumbers is given in the following example.

Each component of the file *records* contains an employee's telephone number and salary. Set the file's access matrix thus:

```
      'records' ⎕tie 202
      ⎕←am← 2 3 ⍴ 3000 0 0 3000 ¯1 161803
 3000      0       0
 3000     ¯1 161803
      am ⎕stac 202
```

In addition, the application designer can provide two locked functions:

```
   ∇ ptie
[1]   'records' ⎕tie 25 161803
   ∇
   ∇ r←pread n
[1]   r←2↑⎕read 25,n,161803
   ∇
```

A user that didn't know the passnumber would have to use these functions, and would not be able to read the salary information. A user that knew the passnumber could read the entire record.

Using this general technique, it is possible to impose quite complex restrictions on file authorization--in fact, any sort of restriction that can be stated as an APL program.

## Chapter 5: COMMUNICATING WITH A SHARP APL/370 SYSTEM

You can use SHARP APL/PC's Asynchronous Communications Auxiliary Processor (ACOM) to access mainframe computers or to drive plotters, printers, or other serial devices through your Personal Computer's serial ports.

The first half of this chapter describes some simple functions for using ACOM. Most functions are listed with comments, but a good understanding of how they operate can only be obtained by experimentation and careful reading of the reference material. These functions do little in the way of checking their arguments.

The second section comprises operating instructions for a more complete set of functions you can use:

- To sign on the I.P. Sharp Online Service;
- To interact with the Online Service, to which your PC appears to be an ordinary APL terminal;
- To transfer objects between active workspaces; and
- To upload and download APL files.

The functions used in this chapter are to be found in two workspaces: 10 *acom* on the I.P. Sharp Online Service and in the distributed workspace *acom*.

For detailed technical information about ACOM, see chapter 12.

## SIMPLE FUNCTIONS FOR USING ACOM

ACOM has six commands:

**Start**
enables the port and initializes communication parameters.

**Transmit/receive**
sends or receives data.

**Transmit break**
clears the ACOM buffer and sends a BREAK signal.

**Stop**
disables the port.

**Inquire**
reports the port's current state.

**Set/read frequency table**
sets or returns the character frequency table used in translation 4.

These commands are given to ACOM as character vectors through a shared variable. Each vector starts with a 16-element header, which specifies the command and its parameters. Transmit/receive may have further elements: the data to be transmitted.

Let's start with a cover function to simplify the first step--sharing two variables with ACOM:

```
    ∇ share
[1]   'invalid offer' ⎕signal(0∈700 ⎕svo 2 3 ⍴'CTLDAT')
      /11
[2]  ⍝1854339pc360:ws:acom:pc:share.ipsa.fun.2.27714.
    ∇
```

The variable *CTL* is used for the control commands: Start, Transmit break, Stop, Inquire, and Set/read frequency table. *DAT* is used for the data command: Transmit/receive.

The function *Δdo* sends a command string to ACOM through the appropriate shared variable, and waits for the return string:

```
    ∇ rs←msg Δdo cs
[1]   →(⎕av[1]=cs[0])/s2
[2]  s1:→(⎕av[2]≠(rs←CTL⊣CTL←cs)[4])⍴0 ⋄ msg Δer rs ⋄ →
[3]  s2:→(⎕av[2]≠(rs←DAT⊣DAT←cs)[4])⍴0 ⋄ msg Δer rs ⋄ →
[4]   ⍝
[5]   ⍝ get AP700 to do a command
[6]   ⍝1854339pc360:ws:acom:pc:Δdo.ipsa.fun.2.27738.
    ∇
```

Let's make sure that ACOM is working:

```
      )load acom
saved  1984-04-08 17:08:36

      share

      □←q←Δdo □av[□io+16↑3]
□
```

The direct display of ACOM's response is not very meaningful; more informative is:

```
      □av⍳q
3 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
```

This shows that the Stop command has been issued on port 0. The fifth element is 1, indicating success.

Analysing return codes can be tedious, so here is a function that does it for you (see chapter 12's section on **Return codes**):

```
    ∇ Δrcode2 rc2
[1]   RC2a[□av⍳rc2[0];0]
[2]   ''
[3]   →(rc2[0]≠□av[2])/s2
[4]  s1:RC2b1[□av⍳rc2[1];] ⋄ →s3 ⍝failed
[5]  s2:RC2b2[((8⍴2)⊤□av⍳rc2[1])/⌽⍳8;] ⋄ →s3 ⍝other
[6]  s3:''
[7]  ⍝
[8]  ⍝ analyse Return-Code-2
[9]  ⍝1854339pc360:ws:acom:pc:Δrcode2.ipsa.fun.2.27766.
    ∇
```

The state of the port's modem signals--indicated in the seventh element of every return string--can be displayed by *Δrmodem*:

```
    ∇ Δrmodem ⍵;a
[1]   RM1[a←((8⍴2)⊤□av⍳⍵)/⌽⍳8;] ⍝signals on
[2]   RM2[(~ 5 4 1 0 ∈a)/ 5 4 1 0 ;] ⍝signals off
[3]  ⍝
[4]  ⍝ analyse modem-signals
[5]  ⍝1854339pc360:ws:acom:pc:Δrmodem.ipsa.fun.2.27770.
    ∇
```

The function *Δer* reports errors; it's used by *Δrcode2* and *Δrmodem*:

```
    ∇ msg Δer rs
[1]   'ERROR:' ⋄ msg
[2]   'RETURN-CODE-2:' ⋄ Δrcode2 rs[4 5]
[3]   'MODEM SIGNALS:' ⋄ Δrmodem rs[6]
[4]   pc ⋄ →
[5]  ⍝
[6]  ⍝ error report
[7]  ⍝1854339pc360:ws:acom:pc:Δer.ipsa.fun.2.27742.
    ∇
```

Several global variables in the workspace must be defined before the *acom* functions will work properly. *config* sets them:

```
    ∇ config;a;x;y;z;⎕io;p
[1]   ⎕io←0
[2]  lp:⍞←'port no.? (0/1): ' ⋄ ⎕arbout '' ⋄ ⍞←(,'01'[
      PORT]),⎕av[158] ⋄ x←⍞ ⋄ →((1=⍴x)∧∧/x∈'01')↓lp ⋄
      PORT←⍎x
[3]  lp1:⍞←'speed? (300/1200/2400/4800/9600): ' ⋄
      ⎕arbout '' ⋄ p←⍎(z← 300 1200 2400 4800 9600)[
      SPEED] ⋄ ⍞←p,((⍴p)⍴⎕av[158]) ⋄ →(5>x←z⍳⎕fi ⍞)↓lp1
      ⋄ SPEED←x
[4]   PARITY←2 ⋄ STOPB←1 ⋄ DATAB←7 ⋄ FLOWC←1
[5]  lp10:a←'Type of connection? (0=Hardwired,   1=Simp
      le modem,'
[6]   a←a,CR,'                          2=Hayes Modem, 3=Pub
      lic Network,'
[7]   ⍞←a,CR,'                          4=Hayes to Public Ne
      twork): '
[8]   ⎕arbout '' ⋄ p←⍎CONNECT ⋄ ⍞←p,((⍴p)⍴⎕av[158]) ⋄ x
      ←⍞ ⋄ →((1=⍴x)∧∧/x∈'01234')↓lp10 ⋄ CONNECT←⍎x
[9]   →(~CONNECT∈ 2 4)/lp8
[10] lp6:⍞←'Tone or Pulse dialling? (t/p): ' ⋄ ⎕arbout
      '' ⋄ ⍞←(,1↑HAYES),⎕av[158] ⋄ x←⍞ ⋄ →((1=⍴x)∧∧/x∈'
      tp')↓lp6 ⋄ HAYES←x,1↓HAYES
[11] lp7:⍞←'Phone number?: ' ⋄ ⎕arbout '' ⋄ p←1↓HAYES ⋄
      ⍞←p,((⍴p)⍴⎕av[158]) ⋄ →(∧/((x←⍞)∈'0123456789,()-
      '))↓lp7 ⋄ HAYES←(1↑HAYES),x
[12] lp8:⍞←'Host? (zoo, ips, other or <blank> for none)
      : ' ⋄ ⎕arbout '' ⋄ p←HOST ⋄ ⍞←p,((⍴p)⍴⎕av[158]) ⋄
      →(∧/((x←⍞)∈⎕av[86+⍳26],' '))↓lp8 ⋄ →((x∧.=' ')∨((
      ⍴x)∈ 3 4))↓lp8 ⋄ HOST←(~∧/x=' ')/x
[13] lp9:⍞←'User Number? (or <blank> for none): ' ⋄
      ⎕arbout '' ⋄ p←ACCT ⋄ ⍞←p,((⍴p)⍴⎕av[158]) ⋄ →((∧/
      x=' ')∨∧/((x←⍞)∈⎕av[86+⍳26],'01234567890'))↓lp9 ⋄
      ACCT←(~∧/x=' ')/x
[14]  CONFIG←1 ⍝<config> run
    ∇
```

Let's use *config* to set the globals to use port 0 on a 1200-baud hardwired connection to the I.P. Sharp Network:

```
      config
port no.? (0/1): 0
speed? (300/1200/2400/4800/9600): 1200
Type of connection? (0=Hardwired,   1=Simple modem,
                     2=Hayes Modem, 3=Public Network,
                     4=Hayes to Public Network): 2
Tone or Pulse dialling? (t/p): p
Phone number?: 365-1466
Host? (zoo, ips, other or <blank> for none): ips
User Number? (or <blank> for none): 1234567
```

### Initializing the port (Start)

Function *Δstart* initializes the port:!gs

```
    ∇ Δstart
[1]   ⊣'Δstart failed' Δdo ⎕av[16↑0,PORT, 0 0 ,SPEED,
      PARITY,STOPB,DATAB,FLOWC, 1 1]
[2]  ⍝1854339pc360:ws:acom:pc:Δstart.ipsa.fun.4.27778.D
      .
    ∇
```

An alternative to *config* and *Δstart* is the function *mode ω*, where *ω* is the string you would specify to DOS's MODE command.

### The Break command

A cover function for ACOM's Break command can be written in much the same way as *Δstart*:

```
    ∇ Δbreak
[1]   ⊣'Δbreak failed' Δdo ⎕av[16↑2,PORT]
[2]  ⍝1854339pc360:ws:acom:pc:Δbreak.ipsa.fun.3.27734.D
.
    ∇
```

### Inquiring of and stopping the port

In a similar way, cover functions for ACOM's Stop and Inquire commands can be written.

```
    ∇ Δstop
[1]   ⊣'Δstop failed' Δdo ⎕av[16↑3,PORT]
[2]  ⍝1854339pc360:ws:acom:pc:Δstop.ipsa.fun.3.27782.D.
    ∇

    ∇ r←Δinquire
[1]   r←⎕av⍳16↓'Δinquire failed' Δdo ⎕av[16↑4,PORT]
[2]  ⍝1854339pc360:ws:acom:pc:Δinquire.ipsa.fun.3.27754
      .D.
    ∇
```

### Sending and receiving characters

Sending and receiving characters generally depends on your application. However, you may find these functions useful:

```
    ∇ arbout ω
[1]   ⊣'arbout failed' Δdo ⎕av[16↑1,PORT],⎕av[ω]
[2]  ⍝1854339pc360:ws:acom:pc:arbout.ipsa.fun.2.27626.
    ∇

    ∇ r←arbin ω
[1]   r←⎕av⍳16↓'arbin failed' Δdo ⎕av[16↑1,PORT, 0 0 7
      0 1 0 20 3 0],⎕av[ω]
```

```
[2]  ⍝1854339pc360:ws:acom:pc:arbin.ipsa.fun.2.27622.
     ∇

     ∇ charout ⍵
[1]    ⊣'charout failed' Δdo ⎕av[16↑1,PORT,3],⍵
[2]  ⍝1854339pc360:ws:acom:pc:charout.ipsa.fun.2.27634.
     ∇

     ∇ r←charin ⍵
[1]    r←16↓'charin failed' Δdo ⎕av[16↑1,PORT, 3 0 7 0
       10 0 100 3 0],⍵
[2]  ⍝1854339pc360:ws:acom:pc:charin.ipsa.fun.2.27630.
     ∇
```

## Connecting with the I.P. Sharp Network

The nodes of the I.P. Sharp Network are computers that handle the traffic of many users and several hosts. Dialup ports answer telephone calls (with a high-pitched squeal) and put the node into a listening mode. Hardwired connections are polled periodically to see if anyone is trying to connect. The procedure for the two cases is different, and is illustrated by the functions *Δhardwire* and *Δhayes*:

If you are using a public network, use *Δhayes* or *Δmodem* to connect you the network. Then use *Δtalk* to connect you to the I.P. Sharp Network.

```
     ∇ Δhayes;a;b
[1]  ⍝get phone number from user
[2]    a←HAYES ⋄ →s1
[3]  lp1:⍞←'phone number for hayes modem to dial: ' ⋄
       ⎕arbout '' ⋄ a←⍞
[4]  s1:→(0=⍴a←(' '≠a)/a)/lp1 ⋄ →(~∧/a∈'0123456789,pt -
       ()')/lp1
[5]  ⍝hayes should have config switch 1 UP so that ~DTR
       will drop connection
[6]    Δstop
[7]    ⊣⎕dl 1 ⍝gives hayes chance to see ~DTR
[8]    Δstart
[9]  ⍝make dialup connection
[10]   b←'command to hayes failed' Δdo ⎕av[16↑1,PORT, 3
       0 13 0 0 0 0 3 1],'⍺~e0 d',a,CR ⍝dial command to
       hayes
[11]   →(('∩OTT∈∩~',CR)≡¯8↑b)/0 ⍝dial successful?
[12]   →((,CR)≡16↓b)/0 ⍝assume successful (can't really
       tell)?
[13]   'failed to make dialup connection' ⋄ 'response fr
       om hayes to ATd command:' ⋄ 16↓b ⋄ →
[14] ⍝
[15] ⍝ establish a dialup connection using a hayes smar
       tmodem
[16] ⍝hayes configuration switch 1 must be UP (DTR supp
       orted)
     ∇
```

```
     ∇ Δmodem
[1]    →(0=(Δinguire)[0])ρs1
[2]    Δstop
[3]    ⍝⎕dl 6
[4]    →(0=(Δinguire)[0])ρs1
[5]    ⍝⍞⍝⍞←'DSR high, please check and hit CR'
[6]   s1:
[7]    Δstart
[8]    ⍝⍞⍝⍞←'make dialup connection and hit CR when done
       '
[9]    ⍝
[10]   ⍝ establish dialup connection using phone/modem
[11]   ⍝1854339pc360:ws:acom:pc:Δmodem.ipsa.fun.2.27758.
     ∇

     ∇ Δtalk;a;b
[1]    'on each input either; line to send; CR to receiv
       e, space/CR to exit'
[2]   lp1:⍞←'⎕' ⋄ ⎕arbout '' ⋄ →((,' ')≡a←⍞)/0 ⍝exit?
[3]    →(16=ρb←'Δtalk failed' Δdo ⎕av[16↑1,PORT, 3 0 7 0
        3 0 10 3 0],a,(0≠ρa)/CR)ρlp1
[4]    16↓b ⋄ →lp1
[5]    ⍝
[6]    ⍝ simple terminal emulator
     ∇
```

Δ*talk* allows full-duplex communication with the public network. If Δ*talk* prompts you for input when you expect output, just press <return>.

## Signing on to a SHARP APL/370 system

Once you're connected to the I.P. Sharp Network, you have to select the **host** you want to communicate with. (A host is a SHARP APL/370 system.) Δ*signon* selects the host, prompting you for any information it needs:

```
     ∇ Δsignon;a;b
[1]    a←8
[2]   lp1:b←DAT⍪DAT←⎕av[16↑1,PORT, 0 0 7 0 1 0 5 3 0 1],
       ⎕av[111 127 13] ⍝tx/rx
[3]    →(⎕av[10 10 7]≡¯3↑b)ρs1 ⋄ 'retrying' ⋄ ⍝⎕dl 1 ⋄ →
       (0<a←a-1)/lp1 ⋄ →er1
[4]   s1:
[5]    ⍝
[6]   lp21:⍝DAT⍝DAT←⎕av[16↑1,PORT,3, 0 0 0 0 0 0 0 0 1],
       ')',HOST2,CR ⍝tx 'xxx'
[7]    →(⎕av[8 8 7]≡¯3↑16↓b←DAT⍝DAT←⎕av[16↑1,PORT, 0 0 7
        0 100 0 100 3 0 1])/s4 ⍝rx
[8]    ⎕←arbtype[⎕av⍳16↓b]
[9]   lp22:⍞←'Host? (zoo, ips or other): ' ⋄ ⎕arbout ''
       ⋄ ⍞←HOST2,((ρHOST2)ρ⎕av[158]) ⋄ →(∧/((a←⍞)∈⎕av[86
       +⍳26],' '))↓lp22 ⋄ →((a∧.=' ')∨((ρa)∈ 3 4))↓lp22
       ⋄ HOST2←a ⋄ →lp21
```

```
[10] s4:
[11] ⍝
[12] lp3:⍎DAT←⎕av[16↑1,PORT,3, 0 0 0 10 0 100 0 6],')',
     ACCT2,':',PW,CR
[13]  →((CR,'  ')≡3↑a←16↓DAT)/0 ⋄ ⎕←a,16↓DAT⍎DAT←⎕av[16
     ↑1,PORT, 3 0 0 0 10 0 100 3 0]
[14] lp5:⍞←'User Number? : ' ⋄ ⎕arbout '' ⋄ ⍞←ACCT2,((⍴
     ACCT2)⍴⎕av[158]) ⋄ →(∧/((a←⍞)∈⎕av[86+⍳26],'012345
     67890'))↓lp5 ⋄ ACCT2←' ',a
[15]  'pass word? : ' ⋄ ⎕arbout '' ⋄ PW←⎕av[2↓⎕arbin 3
     22 5 10] ⋄ →lp3
[16] ⍝
[17] er1:'initial connect failed' ⎕er b ⋄ →
[18] ⍝
[19] ⍝ signon to host (assumes connection to ipsanet ha
     s just been established and is in an initial know
      state, and HOST2/ACCT2 are setup)
[20] ⍝1854339pc360:ws:acom:pc:⎕signon.ipsa.fun.5.27774.
     D.
    ∇
```

⎕*signon* assumes that your connection to the I.P. Sharp Network has just been established, and so is in a known state. ⎕*signon* is unlikely to work when your connection is in any other state. Functions ⎕*hardwire*, ⎕*modem*, ⎕*hayes*, ⎕*talk* get the connection to this "known state".

⎕*on* puts together network connection and signon:

```
    ∇ on;PW
[1]   configck ⍝check HOST and USER NUMBER set up
[2]   →(s1,s2,s3,s4,s5)[CONNECT]
[3]  s1:⎕hardwire ⋄ →s9
[4]  s2:⎕modem ⋄ →s9
[5]  s3:⎕hayes ⋄ →s9
[6]  s4:⎕modem ⋄ ⎕talk ⋄ →s9
[7]  s5:⎕hayes ⋄ ⎕talk ⋄ →s9
[8]  s9:
[9]   ⎕signon
[10]  ⎕signal 998 ⍝cut back stack and exacute Wmf
[11] ⍝
[12] ⍝ on host (establish host connection/ttask)
[13] ⍝1854339pc360:ws:acom:pc:on.ipsa.fun.6.27698.
    ∇
```

## Signing off

Function *off* undoes all that *on* does:

```
      ∇ r←off;Wsrmode;Wsnder
[1]   →('P'∈Wsystem)↓l1 ⋄ rxflush ⋄ r←charin ')off',CR
      ⋄ ⎕stop ⋄ r←r,CR,'on PC' ⋄ →0
[2]  l1:Wsnder←0 ⋄ 1 Winits 'off' ⋄ Warbflg←⍴⎕trap←''
    ∇
```

## Improving on these functions

The functions described above have been kept simple to assist in explaining the concepts involved. They can be improved in a number of ways.

Communications, particularly over telephone lines, are prone to error. How do you detect these errors? How do you correct them? ACOM has an integrity checksum (ICS) option that can be selected in the Transmit/Receive command. ACOM can insert checksums on transmissions and use checksums in received data. To operate, this requires a cooperating task at the mainframe end. One partner inserts the checksum, the other uses it. If an error is detected, the relevant message can be retransmitted.

**Caution: When in doubt, examine the return code.** Working with ACOM can produce situations where things have got out of sync, and the buffer may contain unexpected results.

## SHARP APL TRANSFER SOFTWARE

Two workspaces, one on a SHARP APL/370 system and one on a SHARP APL/PC system, together provide a facility for connecting the two systems. Once connected, they provide a reliable means for conducting an interactive APL session or exchanging APL objects between the two systems.

The functions used reside in two workspaces: the *acom* workspace distributed with SHARP APL/PC, and 10 *acom* on the I.P. Sharp Online Service. Copies of 10 *acom* may be obtained free of charge for installation on an in-house SHARP APL/370 system.

Most ACOM cover functions have the same names in the two workspaces. The higher-level functions are identical; the lower-level functions are different. It is assumed that signon is always initiated from the IBM PC.

### Getting started

Once SHARP APL has been brought up on the IBM PC:

```
      )load acom
saved .....
```

When you load *acom*, some variables are initialized, and two variables--*CTL* and *DAT*--are shared with ACOM.

You can alter the communication configuration with *config*: it displays with the existing settings, and prompts you to change them. You make the change permanent by saving the workspace. For example:

```
      config
port no.? (0/1): 0
speed? (300/1200/2400/4800/9600): 1200
Type of connection? (0=Hardwired,   1=Simple modem,
                     2=Hayes Modem, 3=Public Network,
                     4=Hayes to Public Network): 2
Tone or Pulse dialling? (t/p): p
Phone number?: 365-1466
Host? (zoo, ips, other or <blank> for none): ips
User Number? (or <blank> for none): 1234567
```

You sign on a mainframe APL task with *on*>

```
      on
Host? (zoo, ips or other): ips
User Number? : 1234567
pass word? :
```

From this point, all your entries are passed to the mainframe for execution.

ACOM's transfer facilities require that the mainframe task be conditioned. You do this by loading 10 *acom*:

```
      )load 10 acom
saved ....
)term noidles set, ⎕io←0
```

### Switching between mainframe and PC

Functions *mf* and *pc* switch the PC's keyboard between the two systems. *mf* directs keyboard entries to the mainframe task; *pc* directs them to the PC task. Each can be executed on either system, and each prints an appropriate message. For example:

```
      pc
on PC
      ⎕ul
1

      mf
on MF
      ⎕ul
322
```

### Transferring APL objects

You can use *objin* or *objout* to transfer a set of objects from one active workspace to the other. You can use *filein* or *fileout* to transfer the contents of a SHARP APL file from one system to the other. In order to minimize communication errors, all these functions use eight-bit transparent translation mode with checksums added.

#### Exchanges between active workspaces

If you've signed on a mainframe APL task, and loaded 10 *acom*, you can use these functions to exchange APL objects (including functions, variables, and packages):

```
      mf
on MF
      )copy 13 starmap earthview
saved ....
      objout ' earthview '
on PC
transfer completed                (from mainframe to PC)
on MF
      )erase earthview
      pc
on PC
      ⎕nc 'earthview'
3
      objout 'earthview'          (from PC to mainframe)
```

```
on MF
transfer completed
on PC
      )eras earthview
      mf
on MF
      ⎕nc 'earthview'
3
```

*earthview*, a function, is moved from the mainframe to the PC and then back to the mainframe. *objin* and *objout* each takes a namelist as its right argument; the namelist can include functions, variables (both simple and enclosed), and packages. The objects must exist in the source workspace.

Watch out for name conflicts in the receiving workspace. Variables are recreated with assignment from within execute (⍎var←value); functions are defined with 3 ⎕*fd*; and packages use both techniques.

**Do not try to transfer** 10 *acom* **or the PC's** *acom* **workspace.**

**The functions are symmetric:** *objin* and *objout* may be called from either the PC or the mainframe.

### Transferring files

You transfer SHARP APL files much the same way as single objects. The functions used are named *filein* and *fileout*, and can be use symmetrically, from either the PC or the mainframe.

The right argument is the identifier of a SHARP APL file accessible from the source task. On the PC, the file identifier must include the drive number; on the mainframe, this must include the library number. If problems are encountered tying the source file, one of these messages is issued:

*Error: source file does not exist*

*Error: access error tying source file*

The left argument is the identifier of a file to be created on the destination task. This identifier must include the drive number on the PC, or the library number on the mainframe. The file must not already exist; if it does, the following message is displayed, and the transfer is aborted.

*Error: Destination file cannot be created*

For example: file 1234567 *software* is **downloaded**, that is, transferred from the mainframe to the PC; A file named *software* is to be created in the current directory on drive C:

```
      mf
```

```
on MF
      '3 software' fileout '1234567 software'
on PC
transfer completed
on MF
```

### Interrupting this session

When you are in PC mode--that is, when your keyboard entries are executed only by the PC APL task, perhaps because you've executed *pc*--you manually signal input interrupt by pressing <shift ScrollLock>, and attention or interrupt by pressing <Ctrl Break>. However, when you're in MF mode--that is, when your keyboard entries are sent to the mainframe APL task, perhaps because you've executed *mf*--you are under the control of an APL function on the PC that emulates a terminal; to signal input interrupt, or attention, or interrupt manually to the mainframe requires a different process:

To signal input interrupt when the mainframe is awaiting input, press <shift ScrollLock>. You are prompted:

*input interrupt: p=on PC, m=send to MF?*

If you reply *p*, the terminal emulation function (on the PC) is suspended.

If you reply *m*, BACKSPACE is sent to the mainframe function.

To signal attention or interrupt when a mainframe function is running, pressing <Ctrl Break> twice. You are prompted:

*break: p=on PC, m=send to MF?*

If you reply *p*, the terminal emulation function is suspended.

If you reply *m*, attention is signalled to the mainframe task.

### Building other cooperating data transfer applications

*α Winits ω*
sends the expression *ω* (a character vector) to be executed by the other task. *α* indicates which task will transmit first: 0 for this task, 1 for the other.

*Wsnd ω*
sends character vector *ω* to the other task.

*r←Wrcv*
"receives" character vector *r* from the other task.

*r←Wsndrcv ω*
sends character vector *ω* to the other task, and waits for response *r*.

*r*←*Wrcvsnd*
"receives" character vector *r* from the other task, and send a response consisting of the first character of *r* (with the one exception noted below).

Character vector *w* must never be longer than 720 characters. The first character must be: *⎕av[⎕io+4]* for all but the last transmission; and *⎕av[⎕io+3]* for the last transmission

*Wrcvsnd* will **not** send a response if the first character of the received data is *⎕av[⎕io+3]*. This allows the receiving application to wait until the last moment to send a final acknowledgement.

*Wsndrcv*, *Wrcvsnd*, and *Wrcv* retransmit data if a checksum error is detected. A receiving application that detects a checksum error send a response consisting of a special character that is interpreted by the sending application as a request to retransmit the previous message. Applications can alternate sending and receiving in order to handle this possibility.

If one end of the application is interrupted for some reason (for example, by an APL error), you can usually restart by intervening manually (for example, correct the error; *→⎕lc*). If, however, you don't want to continue the application, enter:

```
abort
```

This signals event 900 at both ends.

# Chapter 6: SESSION MANAGER

This chapter describes the concepts and facilities of the SHARP APL/PC Session Manager. The Session Manager forms the interface between you and the SHARP APL/PC system. It receives **input** from the keyboard and from APL; **stores** this input in the session log; and generates **output**, for the Personal Computer's screen and for APL.

The Session Manager forms part of file CP8X.EXE.

For an introduction to the Session Manager, see chapter 3 **Using SHARP APL/PC**.

The Session Manager has many powerful and useful features. One of these features resets all Session Manager states to their defaults. If you find the Session Manager in a condition you neither understand nor want, press <Esc>.

## Session log

Output from APL, and APL input from the keyboard are appended to the **session log** (see figure 6-1). Thus, the session log is a faithful record in the Personal Computer's memory of the transactions between you and APL. No deletion or rewriting is allowed.

The session log is a series of APL **lines**. Each such line is a string of APL characters (including BLANK), ended by the NEWLINE character. Lines received from the keyboard may be any length. Lines from APL are no longer than the value of system variable □*pw* (print width; default in SHARP APL/PC is 80). That is, a single output from APL may require several lines if the output is longer than □*pw*. For further discussion of □*pw* and its effects, see the **SHARP APL Reference Manual**.

A single line in the session log may require several screen rows to be displayed; nonetheless, the Session Manager treats it as one line, and most Session Manager commands are oriented to session log **lines** rather than screen **rows**.

Lines are appended to the log until it is filled. Then, the oldest lines are deleted as new lines are appended. Several lines may be deleted to make room for a long line.

When you start APL, the area of memory available to the session log is set by a CP8X start parameter; the default is 4096 bytes. You can expect 4096 bytes to hold between a hundred and two hundred APL lines. For information about start parameters, see chapter 2 **Getting started**.

Each line in the session log has a unique number associated with it: the first line appended to the log is line 0, the hundredth is line 99, and so on. As old lines are deleted to make room for the new, the first line in the session log will not be line 0.

## Screen areas

The Session Manager defines four areas on the screen (see figures 6-2 and 6-3):

upper window
status line
lower window
input stage

All of these areas need not be in use at once.

The **upper window** shows a portion of the session log. This window is static: it doesn't move over the log unless you move it. The upper window has 0 to 24 rows, always starting at the first screen row. If you don't move the upper window, the oldest lines are shown. When you start APL, the upper window has 0 rows.

The **status line** displays information about the Session Manager. The status line divides the upper window from the lower; if you move the status line down, you're taking screen rows from the lower window and giving them to the upper; if you move the status line up, you're taking screen rows from the upper window and giving them to the lower. For information about the contents of the status line, see **Status line**. When you start APL, the status line is at the top of the screen.

The **lower window** shows a portion of the session log. Transactions normally scroll through the lower window as lines are written to the screen. The lower window has 0 to 24 rows. When you start APL, it has 23 rows.

The **input stage** shows the input line yet to be released. It usually has one row, but will expand to fill the lower window as required. Pressing the <down> key adds blank rows to the input stage; pressing <down> repeatedly could expand the input stage to 24 blank rows. (The maximum size the input line can reach is governed by the I start parameter described in chapter 2.)

## Interpreting keyboard input

When you press any of your PC's keys, the keystrokes are interpreted and acted on. You can:

- Type characters in the input stage, edit them, and release them for processing.
- Recall lines from the session log.
- Control the windows on the monitor.
- Interrupt processing.
- Display and change the Session Manager's state.

Figure 6-1: Session log

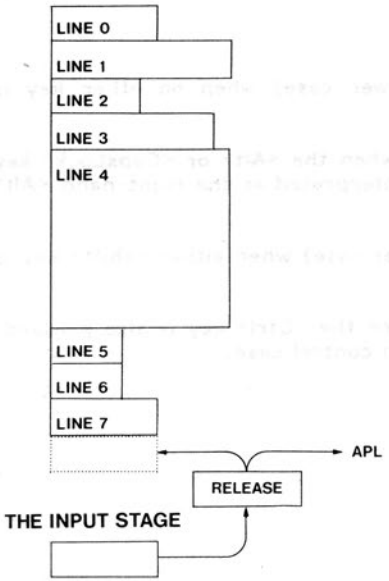



Figure 6-2: Screen areas

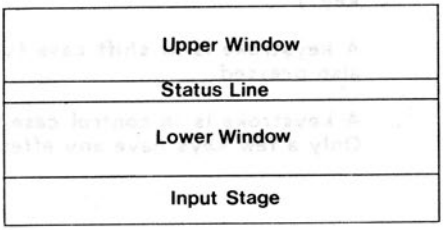
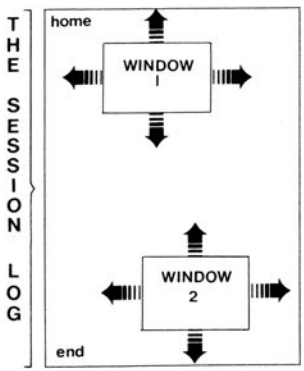



Figure 6-3: Windows



## Cases

Each PC key has a meaning in each of four cases:

base
alternate
shift
control

A keystroke is in **base case** (lower case) when no other key is pressed.

A keystroke is in **alternate case** when the <Alt> or <CapsLock> key is also pressed. (<CapsLock> is interpreted as the right-hand <Alt> key.)

A keystroke is in **shift case** (upper case) when either <shift> key is also pressed.

A keystroke is in **control case** when the <Ctrl> key is also pressed. Only a few keys have any effect in control case.

## KEYBOARD INPUT

To create keyboard input for APL:
- type characters into the **input stage**;
- edit the contents of the stage; and
- **release** the contents of the stage for processing.

### Input stage

The input stage is where you enter characters for input to APL, and edit them. When you release the contents of the stage, the stage is cleared.

The input stage appears at the bottom of the monitor. It expands as to hold what you type (up to the limit established by the I startup parameter).

### Typing characters

You type characters into the stage by pressing any of the PC's usual character keys, in base, alternate, or shift case. These are the keys shown in figure 6-4. <PrtSc> in base case elicits *.

The SHARP APL/PC keyboard is a **union keyboard**: it unites the common APL keyboard with the American National Standard keyboard (such as the PC's). Where an ANS character is similar to an APL character--such as parentheses, brackets, semicolon, and colon--the ANS positioning takes precedence.

For a discussion of the union keyboard's principles, see appendix C **Union keyboard and union character set**. For a summary of the keyboard, see appendix A **Keyboard and character set**.

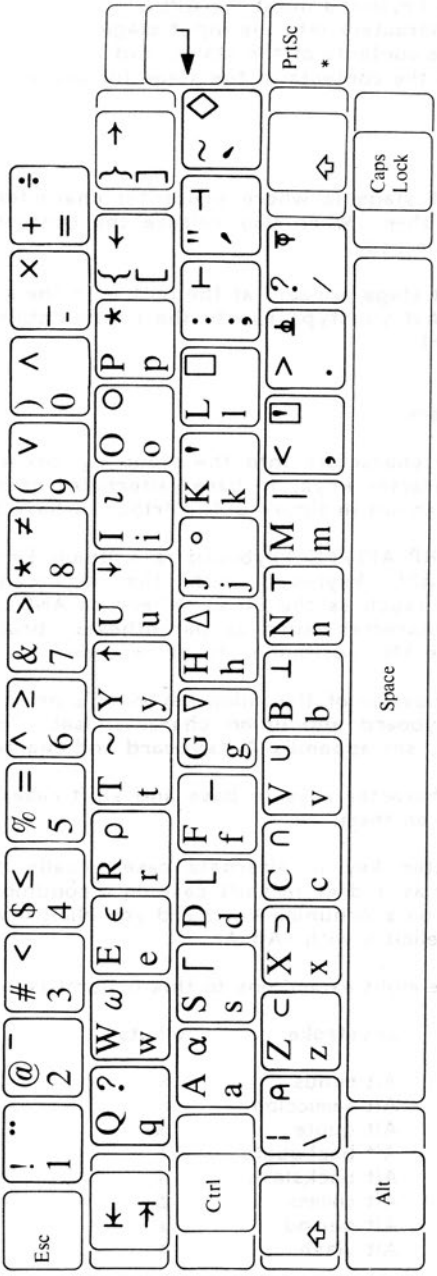
The 46 character keys in base and shift cases elicit the characters engraved on them.

A character key in alternate case usually elicits the same APL character as it does in shift case on a nonunion APL keyboard. For example, on a nonunion keyboard you elicit ⍺ with <shift A>; on the PC, you elicit ⍺ with <Alt A>.

There are eight exceptions to this general rule:

| keystroke | elicits |
|---|---|
| Alt minus | × |
| Alt semicolon | ⊢ |
| Alt quote | ⊣ |
| Alt backquote | ⋄ |
| Alt backslash | ⍎ |
| Alt comma | ⍞ |
| Alt period | ⍬ |
| Alt slash | ⍀ |

**Figure 6-4: Main keyboard**



The alternate-case characters may be shown by stickers on the fronts of the keys; these stickers are supplied with SHARP APL/PC.

Because the keyboard unites the APL and ANS keyboards, some characters are elicited by two or three different keys.

**Key names.** In this handbook, some keys are called by English words rather than by their symbols. Table 0-1 (in the section **About this book** after the table of contents) gives the key number, North American keytop marking, and handbook name. (Every key has a key number; see figure B-3 in appendix B.)

### Overstriking

<Alt backspace> backspaces the cursor, and allows you to overstrike the character there. This is **overstrike mode**, and is indicated by the cursor, which changes from the blinking underbar to a blinking block.

For example, to form the composite APL character ⌽, you type <Alt O> <Alt backspace> <Alt M>.

If you try to form an illegal overstrike (for example, ○ overstruck by /), the latter character is rejected (it doesn't appear on the monitor), and the blinking-block cursor remains stationed over the first character. Because <space> is always a legal overstrike, you can exit overstrike mode by pressing <space>.

For a list of legal overstrikes, see table A-2 in appendix A. The order of overstriking doesn't matter: ○ | and | ○ are equivalent.

### Caps reversal

You turn caps reversal on by pressing <Ctrl Alt> or <Ctrl CapsLock>; when you do, "CAPS" appears in the status line.

Caps reversal reverses the base and shift cases for the 26 letter keys; for example, when caps reversal is on, you type *A* by pressing just <A>, and *a* by typing <shift A>.

You turn caps reversal off by pressing <Ctrl Alt> or <Ctrl CapsLock> again.

### Numeric lock

You activate the numeric keypad by pressing the <NumLock> key; when you do, "NP" appears in the status line.

The numeric keypad comprises only those keys on the far right of the keyboard marked:

0 1 2 3 4 5 6 7 8 9 . + -

It does not include <NumLock>, <ScrollLock>, or any other key.

When the numeric keypad is activated, its keys elicit characters in **all four cases**:

- the keys marked with numerals elicit those numerals.
- <Ins> elicits a period.
- <keypad-minus> elicits APL's ¯ 'overbar' (sometimes called 'high minus').
- <keypad-plus> elicits the BLANK character.

You deactivate the numeric keypad by pressing <NumLock> again.

### Editing input

You edit the input stage's contents with these keys in the control keypad:

<backspace> or <shift backspace>
moves the cursor left one column, destroying the character that was there. (<Alt backspace> is used to form overstrikes.)

<Ins>
toggles insert/replace mode. When you start APL, the keyboard is in **replace mode**: characters you type replace the characters at the cursor. Replace mode is indicated by the absence of ∧ in the status line.

When you first press <Ins>, the keyboard goes into **insert mode**: characters you type are inserted into the stage at the cursor. Insert mode is indicated by ∧ in the status line.

Insert mode is ended and the keyboard reverts to replace mode when you press <return> or <Ins>.

<Del>
deletes the character at the cursor.

<Alt Del>
deletes the input line from the cursor to the last character typed in the line.

<shift Del>
deletes the input line from the cursor to the beginning of the input stage.

<Ctrl Del>
deletes the typeahead buffer; that is, the entries that you've released but that haven't been processed yet. (For information about typeahead, see **Releasing the input stage.**)

### Moving the cursor

You press the <Home> or <End> key to move the cursor quickly to either end of the input stage:

<Home>
moves the cursor to the beginning of the input stage.

<End>
moves the cursor to the last character typed.

You use the PC's familiar cursor control keys (the keys in the control keypad marked with arrows) to move the cursor in the cardinal directions:

<up>
moves the cursor up one row.

<left>
moves the cursor left one column.

<right>
moves the cursor right one column.

<down>
moves the cursor down one row.

You move the cursor quickly to the left or right with the <tab> key (this is the key marked by arrows at the left of the keyboard, directly below the <Esc> key). Although its action is similar to a typewriter's tabulation, SHARP APL/PC's <tab> key moves the cursor eight columns from its current position, rather than to a tab.

<tab>
moves the cursor right eight columns.

<shift tab>
moves the cursor left eight columns.

### Releasing the input stage

Pressing the <return> key in base, alternate, or shift case **releases** the contents of the input stage for processing, and clears the input stage.

APL may not be ready to accept input for processing. If "IN" appears on the status line, the released entry will be processed immediately. If "R" or "C" appears, the entry waits for processing in the **typeahead buffer**. As APL accepts input, the oldest line is removed from the buffer. The number of entries awaiting processing is shown in the status line beside "R" or "C"; * indicates that there are ten or more. (For more information, see **Status line**, below.) Press <Ctrl Del> to delete the typeahead buffer.

When APL is ready to accept the next input, the topmost entry is removed from the typeahead buffer, and sent both to APL and to the session log.

## USING THE PEG

The **peg** marks an APL line in the session log by showing it in reverse video on the screen. The peg appears in either window, and can be moved up or down. The status line shows the line number of the session log line the peg is currently on. The peg is stationary with respect to the screen; so when the window is moved over the log, the peg moves with the window. The peg is invisible when it's on the status line (which is its default location).

### Moving the peg

You move the peg with the <PgUp> and <PgDn> keys on the control keypad:

<PgUp>
moves the peg up one line.

<PgDn>
moves the peg down one line.

### Grabbing a line

You can grab a line displayed on the monitor, and bring it into the input stage for editing and release.

The line to be grabbed is marked by the **peg**. The **pegged line**--the line to be grabbed--is reverse video.

You grab the pegged line with either <keypad-minus> or <keypad-plus>:

<keypad-minus>
completely replaces whatever's in the input stage with the entire pegged line.

<keypad-plus>
inserts the entire pegged line (including any leading blanks) at the cursor (under control of insert/replace mode).

### Setting the reference line

You can mark the current location of the peg in the session log as the **reference line**, that is, a line in the log to which you can return quickly.

<shift keypad-5>
sets the reference line at the current peg.

## SCROLLING THE WINDOWS

You can **scroll** the current window over the session log. This means that you can view different portions of the log than you see by default. You can scroll the window vertically or horizontally.

### Selecting the current window

The current window is the one on which scrolling actions take place. Which window is current is shown by an up or down arrow in the status line. When you start APL, the lower window is current.

<Alt keypad-5>
selects the other window. When you first press <Alt keypad-5>, the upper window becomes current. The current window is indicated by the arrow on the status line.

### Vertical scrolling

<Alt up>
moves the current window up one row, that is, toward older lines in the session log.

<Alt down>
moves the current window down one row, that is, toward newer lines in the session log.

<shift up>
moves the current window up half its height.

<shift down>
moves the current window down half its height.

<Alt Home>
moves the current window to show the oldest line in the session log.

<shift Home>
moves the current window to show the reference line. (The pegged line becomes the reference line when you press <shift keypad-5>.)

<shift End>
moves the current window to show the last line in the session log.

<ScrollLock>
locks/unlocks vertical scrolling of the lower window. Default: unlocked. When you scroll the lower window away from the end of the session log, you can lock the window's position by pressing <ScrollLock>. Otherwise, the lower window will automatically return to the last line when a new last line is put in the session log.

## Horizontal scrolling

Horizontal scrolling is only meaningful when the current window is not in wrap mode. Wrap mode is the default.

When wrap mode is in effect, all displayed lines are wrapped around the screen so that there are no unshown columns. When a window is not in wrap mode, the displayed lines may have unshown columns to the left or the right. The wrap state of the windows is given in the status line; see **Status line**, below.

<Alt NumLock>
enables/disables wrapping of lines in the current window.

<Alt left>
moves the current window left one column.

<Alt right>
moves the current window right one column.

<shift left>
moves the current window left twenty columns.

<shift right>
moves the current window right twenty columns.

## STATUS LINE

This is what the status line shows, left to right:

- The location of the peg:

| value | meaning |
|---|---|
| OFFS | off screen |
| nnnn | a line number in the session log |
| STAT | the status line (default) |
| INPW | the input stage |

- The wrap state of the upper window; see the next item.

- The wrap state of the lower window:

| value | meaning |
|---|---|
| WRAP | lines are displayed on as many screen rows as are required (default) |
| nnn | there are **nnn** columns invisible to the left of the window |
| nnn* | there are **nnn** columns invisible to the left of the window, and columns are invisible to the right (indicated by *). |

- Current window indicator:

| value | meaning |
|---|---|
| ↑ | upper window |
| ↓ | lower window (default) |

- Keyboard state indicator:

| value | meaning |
|---|---|
| IN | input mode |
| R | APL is running |
| C | APL is curbed |
| Rn or Cn | how many typeahead lines are awaiting input processing |
| R* or C* | more than nine typeahead lines are awaiting input processing |

- Caps reversal indicator: "CAPS" if on.

- Insert/replace mode indicator: ∧ if insert mode; blank if replace mode (default).

- Numeric keypad indicator: "NP" if numeric keypad activated.

- Curb indicator: "CURB" if program curb requested; see the discussion of <Alt ScrollLock> in **Interrupting processing**.

- Lower window scroll state:

| value | meaning |
|---|---|
| SC | lower window's bottom row scrolled away from last line in log; will return to last line when a new last line forces scrolling |
| LK | lower window locked, that is, new lines do not appear |

### Moving the status line

You can move the status line up or down; this also closes or opens the upper window:

<Alt PgUp>
moves the status line up one row, thus closing the upper window by one row.

<Alt PgDn>
moves the status line down one row, thus taking one row from the lower window and giving it to the upper.

### Default state

When you start SHARP APL/PC or press <Esc>, this is how the Session Manager is set up:

- The status line is displayed on the topmost row of the screen.
- The lower window takes up all rows of the screen but the topmost.
- The lower window shows the newest input and output lines; it scrolls down, over the log, as new input or output lines are made.
- The upper window (if you make it visible) shows the oldest lines in the log.
- No reference line is set.
- The peg is on the status line.
- Lines in both windows are wrapped.
- The lower window is the current window.
- Caps reversal is off.
- The keyboard is in replace mode.
- The numeric keypad is not activated.
- Output is not curbed.
- The alarm bell is disabled.

### Changing the states

You change the Session Manager's states with these keys:

<Esc>
resets all Session Manager states; see **Default state** above.

<ScrollLock>
locks/unlocks lower-window scrolling. Default: unlocked.

<Alt NumLock>
disables/enables wrapping in the current window. Default: enabled.

<shift keypad-5>
makes the other window current. Default: lower window.

<Ctrl Alt> or <Ctrl CapsLock>
toggles caps reversal. When caps reversal is on, "CAPS" appears in the status line. Default: off.

<Ins>
toggles insert/replace mode. Default: replace mode.

<NumLock>
activates/deactivates the numeric keypad. When the numeric keypad is activated, "NP" appears in the status line. Default: unactivated.

<Alt ScrollLock>
requests a curb, or withdraws the request. Default: no curb requested.

<shift NumLock>
enables/disables the keyboard alarm bell. Default: disabled.

## INTERRUPTING PROCESSING

<Ctrl NumLock>
causes the entire system, APL, and all APs, to stop. At this point, <NumLock> has no effect in any case. To restart processing, press any key but <NumLock>. The key you press has no other effect, except <Break>, which also signals attention to APL.

<Alt ScrollLock>
requests an output curb, or withdraws such a request. When you first press this, "CURB" appears in the status line to indicate that a curb is requested; "IN" or "R" continues to appear as the state. When APL tries to emit output, the output is curbed: APL stops, the output is not emitted, and "C" appears as the keyboard state. APL is uncurbed when you press <Alt ScrollLock> again; "CURB" disappears, and "C" is replaced by "R" or "IN".

<shift ScrollLock>
signals input interrupt (event 1004) to APL (similar to SHARP APL/370's *o* BACKSPACE *u* BACKSPACE *t*).

<Break> (<Ctrl ScrollLock>)
signals attention (event 1002) to APL. If you press this twice, you may signal a strong interrupt (event 1003) to APL.

<shift PrtSc>
causes an attempt to print what's now on the screen. If no printer is attached to the PC, the PC hangs for less than a minute.

<Ctrl Alt Del>
restarts the operating system. All unsaved work is lost. Tied and opened files may be damaged.

## USING ⎕*arbin* AND ⎕*arbout*

You use the ⎕*arbin* and ⎕*arbout* system functions to communicate with the Session Manager.

In SHARP APL/370, you use ⎕*arbin* or ⎕*arbout* to send arbitrary data to a terminal--to have the terminal perform a special action, or to control a printer or plotter.

In SHARP APL/PC, you use ⎕*arbin* or ⎕*arbout* to send arbitrary data to the Session Manager. In particular, you can: make keyboard input invisible; define function keys; and change some control tables.

Arbitrary output codes that perform certain actions on one device will not necessarily perform those--or any--actions on another device. Codes familiar to timesharing users--such as ⎕*arbout* 7 to ring the bell--will not work on the PC. Codes that do work are described in this chapter.

All discussion of ⎕*arbin and* ⎕*arbout* assumes an origin of 0.

Making keyboard input invisible and defining function keys are discussed later in this chapter. For information about changing Session Manager tables, see appendix B **Customizing the keyboard and character set.**

⎕*arbin* or ⎕*arbout* are identical in use, except that ⎕*arbin* returns a result. The rest of this chapter uses ⎕*arbin*.

### The ⎕*arbin* or ⎕*arbout* argument

⎕*arbin* takes an integer or character argument:

```
⎕arbin 3 22 5 ,cmd,scmd,data
```

or:

```
⎕arbin ⎕av[3 22 5 ,cmd,scmd,data]
```

3 22 5
: is the header. It was chosen because it is an unlikely combination of arbitrary output codes for devices and session managers other than the SHARP APL/PC Session Manager.

**cmd**
: is a single integer indicating the Session Manager command being issued.

**scmd**
: is a single integer indicating the subcommand being issued. Some commands don't require this value.

**data**
is any data required by the command. Some commands don't require data.

For a summary of the Session Manager's commands and subcommands, see appendix B **Customizing the keyboard and character set**.

## The □*arbin* result

□*arbin* returns a vector of two or more integers, in this form:

rc sc data

**rc**
is the return code. It is one of these:

| | |
|---|---|
| 0 | no error |
| 1 | header invalid |
| 2 | command invalid |
| 3 | subcommand invalid |
| 4 | length error |
| 5 | domain error |
| 6 | hardware error |

**sc**
is the return subcode. It indicates where in the argument an error was detected.

## MAKING KEYBOARD INPUT INVISIBLE

You can use ⎕*arbin* to make a keyboard entry invisible. An invisible entry does not appear in the input stage as you type it, nor is it recorded in the session log. It is, however, sent to APL.

This feature is similar to the blot facility of SHARP APL/370, and is used for similar things, such as entering a password or passnumber.

The command code for invisible-input mode is 10. ⎕*arbin* ⎕*av*[3 22 5 10] starts invisible-input mode for the following keystrokes, until the entry is released.

This functions prompts for, receives, and returns invisible input:

```
    ∇ r←Δsecret ω
[1]   ⍝ prints ω
[2]   ⍝ and collects input in invisible-input mode.
[3]    r←⎕av[⎕io+2↓⎕arbin 3 22 5 10]⊣⎕←ω
    ∇
```

## DEFINING FUNCTION KEYS

You can program your PC's function keys (the keys marked <F1> through <F10>) to emit keystrokes as if those keystrokes had been pressed on the keyboard. You can define each function key in base, alternate, or shift case.

You define a function key to include **text** and **editing actions** in any combination. The text is APL characters.

You can define a function key to contain several input lines: text 1, release, text 2, release, and so on. However, be careful with the release action. Once an entry is released, it can't be recaptured. If your finger slips and you accidentally press a key that just puts text into the input stage, no harm is done. If, however, the key you accidentally press loads a workspace, much damage can be done--you lose all your unsaved work. Defining your function keys to require your separately pressing <return> is a small price to pay for avoiding a lot of trouble.

You set or read function key definitions with the cover functions in distributed workspace *fkeys* or with the ⎕*arbin* system function.

The function key definition area is limited to six hundred characters. Duplicate definitions for a single function key are accepted and stored although only the first one is used.

### Distributed workspace *fkeys*

You can use the functions in distributed workspace *fkeys* to set or read the function key definitions.

Function *setfkeys* sets the key definitions. Its argument is a character vector containing the text to be emitted, and keywords to identify key numbers and editing actions. All keywords are preceded by a keyword delimiter. A keynumber keyword precedes a key's text and is separated from it by a single blank.

| keywords | keystroke |
|---|---|
| *f*1 through *f*10 | <F1> through <F10> |
| *f*11 through *f*20 | <shift F1> through <shift F10> |
| *f*21 through *f*30 | <Alt F1> through <Alt F10> |

For example, to place the text "*This is text for F*1" into the base case of key <F1>, execute:

```
setfkeys '∘f1 This is text for F1'
```

Similarly, to set text for <F2>:

```
setfkeys  '-f2 More text in F2'
```

In the first example, ∘ is the delimiter; in the second, -. You can set several keys with a single use of *setfkeys* by catenating the

definitions. If you do this, all the delimiters must be the same; for example:

```
setfkeys '|f12 a←12|f20 )load myws'
```

In all of the above examples, the text is displayed when you press the function key, and the cursor is at the end of the text. You use keywords for editing actions to cause the text to be released, and the cursor to be controlled. These are the keywords for editing actions:

*rel*
<return>: release input line for processing.

*delh*
<shift Del>: delete from the cursor to the beginning (home) of the input stage.

*dele*
<Alt Del>: delete from the cursor to the end of the input stage.

*rep*
replace the input stage including the current prompt, or upcoming prompt if typeahead was used.

*home*
<Home>: move cursor to the beginning of the input stage.

*end*
<End>: move cursor to the end of the input stage.

Each editing-action keyword is preceded by a delimiter. For example, to assign text to <shift F2> and cause it to be released, execute:

```
setfkeys '∘f12 Enter this text∘rel'
```

To define a pair of function keys that move the cursor to the beginning of the input stage, and erase all text in the stage:

```
setfkeys '∘f4∘home∘f5∘dele'
```

If you use the delimiter in front of an invalid keyword, the invalid keyword is treated as text. For example, `setfkeys '+f2 2+2'` results in the text 2+2 being defined to <F2>.

Each use of *setfkeys* resets all function keys, so make sure that you include a definition for each key that you want.

You use *getrawfkeys* and *setrawfkeys* to read or set the function keys in the raw form described above.

Function *fmtrawfkeys* takes a raw key definition and formats it into the form used by *setfkeys*. You can keep existing key definitions by catenating the old definitions to the new, and passing this catenation as the argument to *setfk*; for example:

```
q←'|' fmtrawfkeys getrawfkeys
setfk '|f14 a←12|f23 b←14',q
```

The left argument to *fmtrawfkeys* is your choice of delimiter; if you elide it, one is chosen for you from this set:

```
| α ω ∈ - × ⍞ ∇
```

The first one not in your text is used. If they're all in your text, ∇ is used.

### Using ⎕*arbin* for function keys

Using ⎕*arbin*, all definitions are read or set in a single operation. It is not possible to read or set individual key definitions.

Characters in the key definition are indicated by their indices in the result of ⎕*av*. Origin 0 is used throughout this section.

You indicate the beginning of a definition with ⎕*av*[1]; the next character indicates the key to be defined. Valid key values range from 128 through 157 (⎕*av*[128] through ⎕*av*[157]). Value 128 corresponds to key <F1>, 138 to <shift F1>, and 148 to <Alt F1>.

You can define keys in any order. If a key is defined twice, only the first definition is effective.

Editing actions are indicated by ⎕*av*[0] followed by the value for the desired editing action. The editing actions supported are:

⎕*av*[64]
<return>: release input line for processing.

⎕*av*[70]
<Home>: move cursor to the beginning of the input stage.

⎕*av*[71]
<End>: move cursor to the end of the input stage.

⎕*av*[77]
<Alt Del>: erase from the cursor to the end of the input stage.

⎕*av*[82]
<shift Del>: erase from the cursor to the beginning of the input stage.

⎕*av*[87]
replace the input stage, including the current prompt, or upcoming prompt if typeahead was used.

The following example--which you can try at your PC--illustrates the function key facilities:

```
⎕io←0                    Work in origin 0.
```

| | |
|---|---|
| `h←⎕av[3 22 5]` | Header for a Session Manager command. |
| `f←⎕av[6 1]` | 6 is the function key command and 1 is the subcommand indicating set all definitions. |
| `f1←⎕av[1 128]` | 1 is escape for a key definition and 128 is for <F1>. |
| `f4←⎕av[1 131]` | 131 is for <F4>. |
| `e←⎕av[0 64]` | Editing control for release. |
| `c←⎕av[87]` | Editing control to clear the input stage. |
| `⎕arbin h,f,f1,'testing'` | Define the key. |

Experiment with <F1>. See how it follows the rules for typed input. Experiment with insert mode and text and cursor position.

```
⎕arbin h,f,f1,c,'1+1',e,'2+2',f4,'3+3'
```

Experiment with these definitions to see how they work.

Now experiment with your own definitions.

Command 6 subcommand 0 returns the current definitions of the function keys in the `⎕arbin` result. The result is two characters of return code, and then the same data that was used in setting the definitions.

## Chapter 7: WORKSPACES

This chapter discusses SHARP APL/PC workspace identifiers, passwords, and sizes. Aspects of workspaces not specifically discussed here are the same as in SHARP APL/370.

### Identifiers

A **workspace identifier** comprises a drive number and a workspace name.

The **drive number** is a nonnegative integer indicating the disk drive where the workspace was, is, or will be. If it is omitted or 0, the default DOS drive is implied. Workspace operations are always performed in the **current directory** of the specified or default drive.

Drive numbers correspond to DOS drive identifiers as follows:

| Drive number | DOS drive identifier |
|---|---|
| 0 | current |
| 1 | A |
| 2 | B |
| 3 | C |
| 4 | D |
| ... | ... |

The **workspace name** comprises one to eight letters or numerals, beginning with a letter; only the first alphabet--*a b c*, etc.--may be used. If the drive number is present, it is separated from the workspace name by one or more blanks. Leading and trailing blanks are permitted.

These are valid workspace identifiers:

```
'          1 aardvark    '
'a'
'3 a'
'alpha123'
'x9y8z7'
```

Every saved workspace is a DOS file. The DOS filename is formed by padding the workspace name with hyphens to eight characters (if necessary) and adding .SAW to it. Thus, what SHARP APL sees as workspace *zap*, DOS sees as file ZAP-----.SAW. (A SHARP APL workspace's file name is padded with hyphens to avoid any conflict with DOS reserved names, such as PRN.)

### Passwords

A saved workspace in SHARP APL/PC may be protected by a password in just the same way as a workspace in SHARP APL/370. However, the DOS file formed by the workspace is just as accessible to non-APL programs as an unlocked workspace. Similarly, sealing and locking may be circumvented by non-APL programs.

You can regain access to a workspace whose password you've forgotten by setting the H parameter of the CP8X file when you start APL; for details, see chapter 2 **Getting started**.

### Sizes

The size of the active SHARP APL/PC workspace is not fixed. It is a function of available memory on your PC, and your APs and start parameters.

It doesn't matter how big the workspace was the last time it was used: as long as its contents will fit in the currently available space, you can load it. You cannot load a saved workspace whose contents are bigger than available memory; nor can you save a workspace larger than the space available on the drive.

The *)copy* and *)pcopy* commands are far likelier to cause a *ws full* condition on the PC than on a mainframe. When you issue a *)copy* or *)pcopy* command, the entire source workspace is brought into the PC's mainstore. If the whole workspace doesn't fit (together with the active workspace, and all APs), the attempt to copy fails.

## Chapter 8: SHARP APL/PC FILE SYSTEM REFERENCE

The SHARP APL/PC File System gives you access to SHARP APL files on your Personal Computer's disks. A SHARP APL file is a collection of indexable, variable-length, self-described records, each of which is a SHARP APL data object. You store or retrieve data and programs from SHARP APL files with executable SHARP APL statements.

### WARNING

**Untie all files on a diskette before you remove it from the drive.** The PC's memory contains critical information about tied files. If you remove a diskette that holds a tied file, the diskette may be damaged unpredictably. If you mistakenly remove a diskette and then reference it, DOS prompts you with an "Abort, Retry, Ignore" message to give you a chance to recover; you can reinsert the diskette, and reply Retry. If you have already inserted another diskette, or if you insert the wrong diskette after the "Abort" message, the original file on the first diskette may be damaged, and the new diskette may also be damaged. The system command *)off* unties all files.

### File identifiers

A **file identifier** comprises a drive number and a file name.

The **drive number** is a nonnegative integer indicating the disk drive where the file was, is, or will be. If it is omitted or 0, the default DOS drive is implied. File operations are always performed in the **current directory** of the specified or default drive.

Drive numbers correspond to DOS drive identifiers as follows:

| Drive number | DOS drive identifier |
|---|---|
| 0 | default |
| 1 | A |
| 2 | B |
| 3 | C |
| 4 | D |
| ... | ... |

The **file name** comprises one to eight letters or numerals, beginning with a letter; only the first alphabet--*a b c*, etc.--may be used.

If the drive number is present, it is separated from the file name by one or more blanks. Leading and trailing blanks are permitted. These are valid file identifiers:

```
'          1 aardvark   '
'a'
'3 a'
'alpha123'
'x9y8z7'
```

Every SHARP APL/PC file is a DOS file. The DOS filename is formed by padding the file name with hyphens to eight characters (if necessary) and adding .SAF to it. Thus, what SHARP APL sees as file *zap*, DOS sees as file ZAP-----.SAF. (A SHARP APL file's DOS file name is padded with hyphens to avoid any conflict with DOS reserved names, such as PRN.)

For more information about differences between file identifiers in SHARP APL/PC and SHARP APL/370, see appendix D.

### SFS start parameters

**Bn**
specifies the size of the buffer used by the File System, in bytes. **n** may be any value from 2500 to 32767. This buffer is used for all the storage requirements of the File System.

**Hn**
indicates whether the first file tie is to be helped. **n** may be either 0 (specifying that file ties are not be helped) or 1. If you omit H, file ties are not helped.

H1 indicates that the **first** file tie with □*tie* does **not** check the access matrix, and ties the file with full access. Use this to regain access to a file that you are locked out of. When you tie a file this way, you should immediately correct the access matrix. Only the first tie is affected, so you may need to change the P parameter to CP8X to avoid having any ties done before you have a chance to tie the desired file. This facility is analogous to *filehelper* in SHARP APL/370.

**Qn**
indicates whether the File System start message is to be displayed. **n** may be either 0 (specifying that the message is not to be displayed) or 1. If you omit Q, the start message is displayed.

**Tn**
specifies the number of files you can have concurrently tied. **n** may be any integer from 1 to 100. If you omit T, you can have 4 files concurrently tied.

## FILE SYSTEM FUNCTIONS

This section describes each file function, including its syntax. The syntax lines in the descriptions of file functions use the keywords shown below. The actual arguments of file functions can arise from evaluation of any APL expressions. The keywords used in a syntax line show how various portions of the arguments will be interpreted.

**componentnumber**
An integer designating the position of a component in a file.

**drivenumber**
An integer designating a disk drive.

**fileid**
A character vector containing a file name optionally preceded by a drive number.

**filenumber**
An integer designating a file tie number.

**filenumbers**
A vector of zero or more file tie numbers.

**int**
An integer designating the number and position of components to be dropped.

**passno**
An integer passnumber.

**reservation**
An integer scalar (sometimes character-valued) giving maximum file size in bytes.

**result**
The result of a file function.

**value**
Any APL value.

**←**
Indicates that the file function returns a result.

**[ ]**
Encloses optional quantities that may or may not appear in a particular use of a file function.

**' '**
Indicates character-valued information.

*⎕append*
*⎕appendr*

value *⎕append* filenumber [passnumber]
result←value *⎕appendr* filenumber [passnumber]

Creates a new component with a value equal to the left argument and component number one greater than the greatest previous component number. *⎕appendr*'s result is the component number of the appended value.

*⎕avail*

result←*⎕avail* ⍳0

Returns a scalar 1 or 0. 1 indicates that the File System is operating; otherwise, a 0 is returned. A dummy right argument (⍳0) is required but not used.

The File System may be unavailable because you didn't start it before you started APL.

*⎕create*

'fileid [reservation]' *⎕create* filenumber

Creates and ties a file in the current directory of the specified or default drive. The file name must differ from any others in that directory. The right argument designates an available file number.

*⎕drop*

*⎕drop* filenumber int [passnumber]

Drops components from either end of a file. The absolute value of **int** specifies the number of components to drop from the low-numbered end of the file (if **int** is positive) or the high-numbered end of the file (if **int** is negative). If **int** is 0, no components are dropped. Dropping components does not cause the component numbers of remaining components to be changed.

*⎕drop* does not free space on the drive. File space is freed only when the file is erased or copied. For more information, see **Physical storage considerations.**

*⎕erase*

'fileid' *⎕erase* filenumber [passnumber]

Unties the file and deletes it. The left and right arguments must designate the same file.

SHARP APL/PC's □*erase* function uses DOS's CLOSE and ERASE function calls. The file is deleted from the appropriate directory, but its sectors aren't written over; so the contents of an "erased" file are available to a program able to read the bit patterns on the disk.

□*fhold*
□*hold*

[value] □*fhold* value
□*hold* value

Have no effect in a single-user environment.

□*lib*

result←□*lib* drivenumber

Returns a character matrix holding, in each row, the file identifier of a file in the current directory of the designated drive. The result has 22 columns: columns 1 through 10 contain the drive number right-justified with leading blanks; column 11 is blank; and columns 12 through 22 contain the file name left-justified with trailing blanks.

□*names*

result←□*names*

Returns a character matrix holding, in each row, the file identifier of a tied file. The file identifiers are arranged in the order in which they were tied, which is the same order of □*nums*' result. □*names*' result has the same structure as the result of □*lib*.

□*nums*

result←□*nums*

Returns a numeric vector holding the file numbers of all tied files. Each file number is associated with the file identifier in the corresponding row of the result of □*names*.

□*rdac*

result←□*rdac* filenumber [passnumber]

Reads the access matrix of a file, and returns its value as the result.

### □*rdci*

result←□*rdci* filenumber componentnumber [passnumber]

Returns the control information of a component. The result is a three-element vector containing:

- The space (memory) needed to hold the component, in bytes;
- The user number of the task that most recently executed a □*append*, □*appendr*, or □*replace* on the component;
- The time of the most recent □*append*, □*appendr*, or □*replace* on the component, in sixtieths of a second since 00:00 1 March 1960. (This value is only as accurate as the system clock was when the component was appended or replaced.)

Don't assume that the result of □*rdci* is always a three-element vector: future extensions to the File System may require that □*rdci* return a vector of more than three elements.

### □*read*

result←□*read* filenumber componentnumber [passnumber]

Reads a component of a file, and returns its value as the result.

### □*rename*

'fileid [reservation]' □*rename* filenumber [passnumber]

Changes the identifier and reservation of the file designated by the right argument to the values specified by the left argument. The rules governing the left argument are the same as for □*create*, except that the file identifier may be identical to the subject file's current file identifier.

□*rename* does not change the file's ownership to the current user.

You cannot use □*rename* to move a file from one drive to another.

### □*replace*

value □*replace* filenumber componentnumber [passnumber]

Stores the value of the left argument in an existing component.

### □*resize*

reservation □*resize* filenumber [passnumber]

Sets the reservation of the file designated by the right argument to the value of the left argument. The value set is your specification rounded up to the next multiple of 512.

## □*size*

result←□*size* filenumber [passnumber]

Returns a four-element numeric vector containing size information about the file:

- The number of the first component in the file;
- The number of the next component to be appended;
- The physical storage used by the file, in bytes--that is, the file size;
- The physical storage that the file may use.

**Physical storage considerations** discusses the third and fourth elements of the result.

Don't assume that the result of □*size* is always a four-element vector. Future extensions to the File System may require that □*size* return a vector of more than four elements.

## □*stac*

value □*stac* filenumber [passnumber]

Sets the access matrix for a file to the value of the left argument. The left argument must be a three-column integer matrix with nine or fewer rows; the second column of the matrix must be composed of combinations of permission-type values. New access settings are imposed immediately.

## □*stie*
## □*tie*

'fileid' □*stie* filenumber [passnumber]
'fileid' □*tie* filenumber [passnumber]

Ties a file. (In the single-user environment of a PC, □*stie* has the same syntax, effect, and events as □*tie*.) The left argument designates an existing file in the current directory of the specified or default drive. The right argument designates an available file number.

## □*untie*

□*untie* filenumbers

Unties all files designated by the right argument. Elements of the right argument must be distinct, and designate tied files. An empty vector is permitted as an argument, and does not affect any file ties.

## FILE SYSTEM EVENTS

This section describes the events that may be signalled during use of the File System functions that are unique to the File System. For a full description of all APL events (and of APL's event-handling facilities), see the **SHARP APL Reference Manual**.

*file access error*
Access matrix prevents this operation because permission is lacking, or the passnumbers don't match.

*file component damaged*
A software or hardware problem has resulted in a damaged file component.

*file damaged*
A software or hardware problem has resulted in damage to the entire file.

*file full*
The new component was not appended or replaced because there's no room in the file's reservation; or the new reservation is smaller than the file size.

*file index error*
The specified component number is invalid, because it does not exist or because it would exceed 2147483647.

*file name error*
Identified file duplicates an existing file, does not match the tie number, or is not found in the current directory on the drive.

*file quota used up*
User has tried to exceed the maximum number of DOS files that may be created in a directory or the disk is full. This limit is set by DOS.

*file system error*
There is something seriously wrong with the File System. Restart APL.

*file system no space*
The disk is full.

*file system not available*
The File System was not started.

*file tied*
You already have the specified file tied.

*file tie error*
File number not tied to a file; or file number duplicated.

*file tie quota used up*
The user has tried to exceed the maximum number of file ties that may be concurrently established. This limit is set by SFS's T start parameter; the default is 4.

## PHYSICAL STORAGE CONSIDERATIONS

The characteristics of the storage devices used to hold files, and the storage allocation strategies used by the File System complicate the relationship between component size and file size. The average number of bytes of file space required to hold an APL value that occupies *n* bytes of workspace storage is `1.04×(20+n)`.

File storage amounts are always multiples of 512 bytes--the physical record size used by the SHARP APL/PC File System.

The total storage occupied by a file may be as much as several thousand bytes more than this formula would indicate. As components are appended to a file, the storage used grows unpredictably; appending an empty vector may add many bytes, while a subsequent appending of a hundred-element vector may appear to require no additional storage. The reason is that when a file has occupied all allocated storage, appending even a small component causes the allocation of at least an entire physical record.

When components are appended, several small components may share a single physical record. When a *⎕replace* function increases the size of a component, however, the additional physical records that may be needed are not shared with any other component. The space so allocated may be more than what is needed for the component's new value. Additional *⎕replace*s of that component will use that excess space before any further records are allocated.

Suppose, for example, that a file has been built by appending twenty 10-element integer vectors. The space occupied by the file is 2 560 bytes. Now, if each component is replaced by an 11-element integer vector, the occupied space becomes 12 800 bytes because a new physical record has been allocated for each component.

Storage for the file in this example will **not** decrease if components are now reduced again to 10-element vectors--the additional records that were allocated remain allocated.

*⎕drop* does not free any space on a disk.

To reclaim space, you can use a simple utility function like this:

```
    ∇ z←fsqueeze ω;i;n;t
[1]   ⍝ reclaims unused space for file tied to ω
[2]   i←0
[3]   n←1↑1↓z←⎕size ω
[4]   z←1↑2↓z
[5]   'ZZtarget' ⎕create t←((⍳⍴t)∈t←⎕nums)⍳0
[6]  l0:→(n≤i←i+1)/l1
[7]   (⎕read ω,i) ⎕append t
[8]   →l0
[9]  l1:(⎕rdac ω) ⎕stac t
[10]    (n←8↑¯11↑⎕names[⎕nums⍳ω;]) ⎕erase ω
[11]  n ⎕rename t
[12]  ⎕untie t
```

```
[13]   n ⎕tie ⍵
[14]   z←z-1↑2↓⎕size ⍵
       ∇
```

This function recreates the file tied to *⍵*, and returns the space freed. It may change the data's owner or drive.

(*fsqueeze* assumes a 100KB reservation, the default drive, and that components begin with 1 and are all to be copied.)

## SUMMARY OF FILE FUNCTIONS

| Function | Syntax | Permission |
|---|---|---|
| □*append* | expr □*append* fn,pn | 8 |
| □*appendr* | r←expr □*appendr* fn,pn | 16384 |
| □*avail* | r←□*avail* ⍳0 | |
| □*create* | fnm □*create* fn | |
| □*drop* | □*drop* fn,n,pn | 32 |
| □*erase* | fnm □*erase* fn,pn | 4 |
| □*fhold* | □*fhold* svm | 2048 |
| □*hold* | □*hold* fn(s) | 64 |
| □*lib* | r←□*lib* driveno | |
| □*names* | r←□*names* | |
| □*nums* | r←□*nums* | |
| □*rdac* | r←□*rdac* fn,pn | 4096 256 |
| □*rdci* | r←□*rdci* fn,cn,pn | 512 |
| □*read* | r←□*read* fn,cn,pn | 1 |
| □*rename* | fnm □*rename* fn,pn | 128 |
| □*replace* | expr □*replace* fn,cn,pn | 16 |
| □*resize* | p □*resize* fn,pn | 1024 |
| □*size* | r←□*size* fn,pn | 1 32768 |
| □*stac* | mat □*stac* fn,pn | 8192 256 |
| □*stie* | fnm □*stie* fn,pn | |
| □*tie* | fnm □*tie* fn,pn | 2 |
| □*untie* | □*untie* fn(s) | |

| | |
|---|---|
| **cn** | Component number |
| **driveno** | Drive number |
| **expr** | Data |
| **fn** | File number |
| **fnm** | File name |
| **mat** | 3-column access matrix |
| **n** | Integer |
| **p** | Positive integer |
| **pn** | Passnumber or ⍳0 |
| **r** | Result |
| **svm** | Scalar, vector, or 2-row matrix of integers |

Complete access to a file can be given by a permission code of ¯1.

# Chapter 9: SHARED VARIABLES AND AUXILIARY PROCESSORS

This chapter discusses shared variables, the Shared Variable Processor (SVP), and how shared variables permit APL functions to make use of auxiliary processors.

## Shared variables: a discipline of communication

A **shared variable** is one whose value can be set and can be used in two different contexts simultaneously. (In APL terms, such a context is a workspace. But a variable can be shared with a program outside APL, where "workspace" doesn't apply.) Two otherwise independent programs, one in each of those two contexts, can interact; when one of them sets a new value for the variable they share, that new value is visible in the other context. The variable that they share is like a window between them; indeed, an **interface** can be defined as a variable which exists in two different programs. There's a simple but rigorous set of rules governing communication by shared variables. The aim of this chapter is to introduce those rules, together with the APL functions that embody them.

Independent but concurrently running modules are linked by sharing one or more variables between them. Each of the communicating tasks starts by offering to share one or more of its variables with the other task. When each has made that offer to the other, the variable is **shared** between them. Now, when one of them gives a new value to the shared variable (that is, **sets** it), the other has only to **use** it to become aware of its new value. For example, from an APL workspace, a transaction might look like this:

```
x←'command'      Set the variable.
result←x         Use the variable.
```

When *x* is shared, using *x* in the second statement may reveal not the command written there in the first statement, but a quite new value--perhaps the result of executing the command--supplied by the processor with which *x* is shared.

## The Shared Variable Processor (SVP)

The **Shared Variable Processor** is what makes it possible for the same variable to appear in two otherwise separate contexts. The SVP is a separate module of the APL system, and must be started just before you start APL. (See chapter 2 for a discussion of how to start APL and the SVP.)

The SVP manages all use of shared variables. It maintains a table of the names of variables that are shared, and the auxiliary processors which are sharing them with the APL workspace. The SVP manages the mechanism of sharing, including the physical storage and the translation or conversion required when workspaces that are sharing have different conventions for the representation of data. It provides intermediate storage for their values while one partner has set a value but the other has not yet used it. It

enforces the rules that prevent conflict over whose turn it is to set the variable or to use it.

### Sharing with non-APL programs

Sharing can be established between **any** pair of programs that abide by the shared-variable discipline. The APL workspace can share with almost any sort of program capable of running in the same hardware at more or less the same time, even one written in a completely different language. That's how your APL workspace makes use of specialized processors to manage external communications or to control the full screen, and how it sends commands to the operating system to execute DOS commands or to make use of non-APL files.

### Auxiliary processors

An **auxiliary processor** (AP) is a program that provides a service to an APL task but is not itself part of APL, and that uses shared variables to communicate with the APL workspace.

SHARP APL/PC comes with several auxiliary processors, and also permits you to attach others that you may obtain separately or write yourself. Each AP runs in the PC beside SHARP APL, and communicates with SHARP APL through the SVP. You can use the APs supplied with SHARP APL/PC, or you can write an AP to do anything the PC can do. APs expand SHARP APL's scope to particular hardware and software in a simple, consistent way, without requiring that the APL language be redefined or given ad hoc extensions.

The SHARP APL/PC product includes these APs:

- DOS Interface AP (DOSX; described in chapter 11);
- Asynchronous Communications AP (ACOM; which underlies the procedures described in chapter 5, and is described in more detail in chapter 12); and
- Full Screen Manager (FSCR; described in chapter 13).

To use an auxiliary processor, you must start the SVP and the auxiliary processor just before you start APL.

If you use the same APs regularly, you can have your APL.BAT file start them when you start APL. If you start APL without them, the space they would otherwise occupy becomes part of your APL workspace.

## Outline of how variable-sharing works

Each processor (the APL workspace and the AP) keeps its own copy of the shared variable, but marks the variable as "shared". Whenever either of them refers to the shared variable, it checks first with the SVP. The SVP maintains a third place where the value of the shared variable may be stored: three places for one variable. At any particular moment, two of those places contain duplicate values of the shared variable, and the third doesn't matter.

This will become clearer with an example. Suppose you've arranged to share variable *a* in your APL workspace with DOSX, the processor which communicates with the DOS operating system. Now you wish to assign a value to *a* so that DOSX will see it and treat it as a command to be passed to DOS.

- You establish that *a* is shared between your APL workspace and DOSX. (Procedure to do this is described in the next section.) Inside your workspace, the APL system marks the name *a* to show that it refers to a shared variable.

- You assign a value to *a*. When the APL interpreter executes that assignment, it notices that *a* is a shared variable, and notifies the SVP that a new value has been received. The SVP stores its own duplicate of the value of *a*. The value exists both in your workspace and in the SVP's own storage. The SVP notes the you know the latest value of shared variable *a* (since you put it there) but DOSX does not.

- Like most APs, DOSX doesn't do work of its own, but waits for orders from an APL workspace. It sleeps until notified by the SVP that something has happened to affect a variable it is sharing. When DOSX attempts to use the value of the variable that you call *a*, the SVP transfers the duplicate value it has been holding. The SVP notes that both sharers are now aware of the value of *a*, and discards its duplicate. Now the shared variable exists with the same value in your APL workspace and the storage space available to DOSX.

- In due course, DOSX has a reply for you. It sets the value of the shared variable. Any time anyone sets a shared variable, the SVP is notified. The SVP saves a duplicate of the value that DOSX has set. The SVP notes that DOSX knows what the value is (because it set it) but you do not (because you haven't used it yet).

- You use your variable *a* again. (You might be waiting for the reply from DOSX. Or you might be doing something else, and not come back to look at *a* until much later.) When you attempt to use *a*, the SVP copies its duplicate value into your workspace, notes that now both parties are aware of the value of *a*, and discards its duplciate.

That's how the SVP makes *a* seem to exist in both workspaces. The value is not in fact transferred until you attempt to use it, but that

isn't apparent, because any time you ask "what is the value of *a*?", the SVP slips in and delivers the duplicate it has been holding.

### "Sending" and "receiving"

When two programs communicate by sharing a variable, there isn't any special action called "send". Nor is there one called "receive". To send some data to an AP, you use the regular specification arrow ← to assign a value to the variable that you share with the AP. That variable is visible to the AP too, and so you've "sent" the value just by giving it a value in your own workspace. Similarly, to receive data from an AP, you have only to use the value of the shared variable that you find in your own workspace: that's where the SVP delivers it for you.

**The terms "set", "use", and "reference".** A variable is "set" when it occurs immediately to the left of ←, the assignment arrow. Any other occurrence is a "use" of the variable. As will be apparent later, it's also useful to have a term that includes both actions; "reference" means any occurrence of either "set" or "use".

### Establishing a shared-variable link from the APL side

These are the steps that each program that uses shared variables must go through. Your application may embed them in a cover function so that the steps are no longer apparent each time, but they're there. (The corresponding steps taken by a user-written AP are described in chapter 10.)

- Using the system function ⎕*svo* (shared variable offer), you offer to share a variable with an AP. The Shared Variable Processor notes your offer (and gives you a result which means that it has received one offer to share this variable--so far unreciprocated). The SVP transmits your offer to the AP.

- The AP responds with a counteroffer. You can tell it has done so because the result of ⎕*svo* is now 2 when before it was 1. When both sides have made matching offers to share the same variable, sharing is in effect .

- Using the system function ⎕*svc*, you or the AP sets up a **shared variable control**. This prevents misunderstandings that would arise if one partner rushed ahead when it should be waiting for the other.

- Now begins a cycle that may be repeated many times as you work with the AP:

  - You set the shared variable. For example, you give it a value that indicates a command to the AP.

  - The AP uses the value you have set, and does whatever is called for.

- The AP sets the shared variable, either to acknowledge your request, or with the answer to it.

- You use the shared variable to find out what the AP's response was.

- At any point in the process, you can use the function *□svs* to inquire about the **state** of the shared variable. This tells you who doesn't yet know its latest value, so (if the need arises) you can tell whether the AP has read what you sent, or has sent you a reply.

- If you're waiting for a reply from several different shared variables, but you don't know which will reply first, you can have your program wait for a state change by using the state change variable *□sc*. The program waits until the state of one or another of the variables you're sharing changes.

### Establishing and terminating a share

The function *□svo* offers to share one or more variables. The right argument contains the name or names you wish to share. The left argument contains the processor ID (a number) for the AP you wish to share with.

If you use *□svo* monadically (that is, with no left argument), you are inquiring about whether the names at the right have been offered or shared.

**Form of the right argument.** The right argument is a character matrix, with one row for each name. If your offer or inquiry only concerns one name, the argument may be a vector or scalar.

**Surrogate names.** You may find it inconvenient to call your shared variable by the same name that the AP requires. For example, FSCR requires that the variable you share with it have a name that starts with the letters *ctl*. You can make your offer with a **surrogate name**. For example, the offer:

```
124 □svo 'a ctl'
```

offers to the AP whose number is 124 a name which in your workspace appears as *a* but which to the Shared Variable Processor will appear as *ctl*. The rule is that on each row of the right argument there may be either one name or two names separated by blanks; where there are two names, the name on the left is the name in your workspace and the one on the right is the name that will be visible to the AP.

**Restrictions on names:** The name that you offer must be a permissible name for an APL variable; in addition, it can't be longer than 11 characters, and can't be the name of a system variable. It may not have a conflicting use (that is, it may not already have a visible use as a label or as the name of a function). On the other hand, it isn't necessary that it already exist as a variable.

### Duration of sharing

Sharing goes into effect when both you and the AP have made matching offers to share the variable (and the result of *□svo* is 2). Sharing remains in effect until either side retracts the offer. Once you've made a valid offer (and received a result of at least 1), your offer remains in effect until you retract it, implicitly or explicitly.

**Explicit retraction.** You retract a shared variable explicitly by executing the function *□svr*. It takes as its argument the name or names that you wish to retract. For example, if *a* has been shared, *□svr 'a'* retracts the offer. The result returned by *□svr* is the level of sharing that was in effect **before** you retracted.

**Implicit retraction when you wipe out the shared variable.** An offer to share is retracted implicitly whenever the name that you offered ceases to exist. The name ceases to exist as soon as you execute the command *)clear* or *)load*. It also ceases to exist when you expunge it, or when you exit from a function to which it is local.

It is standard practice to write an AP so that, when it finds that you have withdrawn your offer to share, it assumes a normal end to its transactions. It withdraws its counteroffer, and tidies up whatever it was doing. For example, if you have shared a variable with DOSX and used it to open a file, and then you erase your shared variable (or do anything else that terminates the share), DOSX closes the file for you.

**Localization and shared variables.** It's permissible to share a local variable, but if you do the sharing ceases as soon as you exit from the function that localizes it. So you probably don't want to do that unless the function to which your shared variable is local is going to become active rarely enough, or remain active long enough, to make it worth while to reinitiate sharing every time the function is invoked.

When you have shared a global variable *a*, and then, while that share is in effect, you invoke a subfunction that uses *a* as a local name, the local name is the only one visible. Temporarily, your global *a* is **shadowed** by the local meaning. This does not constitute an implicit retraction, and you regain access to the shared variable *a* as soon as the shadowing subfunction ends.

### Clone ID: establishing your exact identity

When you refer to an AP, you identify it by its processor number. Similarly, the AP refers to you by your user number (which on the PC is 3000 unless you've set up the U start parameter differently for the supervisor). On a timeshared system, when you refer to another APL task, you refer to it by its user number.

In the timesharing context it is possible to have more than one task with the **same** user number. Such tasks are differentiated by their **clone** IDs. This needn't concern you in work that's confined to the PC, but it will come up if you move a program to a timeshared

system, and mentioning it now will also explain an otherwise mysterious aspect of shared variable functions.

Your processor ID is really a two-element vector. The first element is your user number. The second element has the value 0 by default. If you use a single number (for example, the number 124 when you offer to share with FSCR), the system understands that as an acceptable shorthand for its full identification, which is 124 0. In the ordinary course of events on the PC, you can forget about that second element.

If you ever have more than one task running and using shared variables at the same time, they have to distinguish themselves by explicitly setting that second element to a positive integer that will be used to tell who's who. That's your clone ID. The function ⎕*svn* sets your clone ID. and its result is the effective clone ID following your proposal. For example, if you want your clone ID to be 99, you might execute *id*←⎕*svn* 99. If your request was proper, your new clone ID is 99, and that is the result returned by ⎕*svn*. (If you've already been using some other clone ID, you can't change it while you already have offers out under the old ID, and the result is the old ID unchanged.)

(You can use ⎕*svn* to inquire what your clone ID is by executing ⎕*svn* ¯1. Since ¯1 isn't a legal clone ID, the system won't set your clone ID to ¯1, but will report what your clone ID actually is.)

Whenever you make an offer to share, your offer implicitly contains your processor ID and clone ID. The expression

```
124 ⎕svo 'ctlS'
```

really means "I, being of known and valid processor ID and clone ID, do hereby offer to you, processor 124 0, my variable named *'ctlS'*."

Suppose you haven't got a valid clone ID and the default clone ID 0 is already being used by another task having the same user number. In that case your attempt to execute any of the shared-variable functions results in the error message *identification in use* (meaning that some other task has clone ID 0 and until you establish a clone ID of your own, any attempt to execute a shared-variable function is ambiguous, and rejected.

### Inquiring about the degree of coupling

The number of partners sharing a variable is its **degree of coupling**. It may be 0, 1, or 2. (Many-way communication among many processors is built up from links each of which has only two ends.)

The monadic form of ⎕*svo* tells you the degree of coupling of the names in its right argument. When the level is 2, sharing is in effect. 1 means the SVP has recorded one offer, but not yet a matching offer. 0 means that no valid offer has been made.

After you've made an offer to an AP, it's prudent to verify that sharing is really in effect (that is, the result of □*svo* is 2) before you charge ahead trying to make use of it. The following is a subfunction that verifies that the name in its right argument really is shared, and, if not, offers to share it with the processor identified in its left argument:

```
    ∇ r←id sharecheck name
[1]   →(2∧.=r←□svo name)/0
[2]   r←id □svo name
[3]   'invalid offer' □signal (0∈r)/11
[4]   r←1 □svc name
[5]  ck:→(2∧.=r←□svo name)/0
[6]   'checking offer to processor ',⍕id
[7]   →□sc/ck
    ∇
```

### Inquiring about offers from others

The function □*svq* inquires about incoming offers addressed to you. Using SHARP APL on the PC, there's no real use for this, since none of the distributed APs ever initiates an offer to share; they just respond to the offers you make to them. But you might choose to write an AP that intiates an offer; in that case, you could detect the offer using □*svq*. □*svq* ⍳0 tells you the processor numbers of those processors that have made offers to you but to which you haven't made matching counteroffers. □*svq* **n** (where **n** is a single ID number) returns a matrix of the names that processor **n** has offered to you. Ordinarily, you'd use □*svq* ⍳0 to find out who's making offers, and then, if the resulting vector of processor numbers is not empty, inquire individually about each processor number in turn.

### Inquiring about the shared variable's state and controls

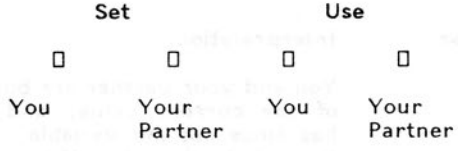
The function □*svs* inquires about the state of the shared variables named in its right argument, and the function □*svc* reports or sets controls. For both functions, the right argument is a matrix, with one name per row; if you're only concerned with a single name, it may be a scalar or vector.

The result of either □*svs* or □*svc* is a boolean matrix with four columns (or a four-element vector, when the argument had rank 1 or 0). The four columns can be considered as two pairs. In each pair, the first describes whoever executed the function, and the second describes the partner. When you use one of these functions in an APL workspace which shares a variable with an AP, then the first member of each pair refers to your APL workspace, and the second member refers to the AP.

**Point of view.** When two processors share a variable, sharing is symmetric. Neither takes precedence over the other. It doesn't matter which made its offer first and which second.

To preserve this symmetry, each of them sees itself as the first member of the pair. So when you're writing code for your own AP, you'll see the result returned from the AP's point of view. In particular, you'll find reference to the AP comes first, followed by the APL workspace with which it is sharing. This is also true in SHARP APL/370, where sharing may be between two APL workspaces. Each sees itself as the first member of the pair, and its partner as the second. Therefore, in this chapter (and in the **SHARP APL Reference Manual**) we refer to **you** and **your partner**, because this way of speaking applies equally well to the APL workspace or the AP.

**"Set" and "use".** Within the result of □*svs* or □*svc*, the first pair of columns concerns **setting** the shared variable, while the latter pair concerns **using** it (that is, reading it to make use of its value). When it's your turn to use the shared variable, you have the right to set its value instead (ignoring the value presumably set there by your partner). This option is discussed in the section **Exceptional routes**, following the full-page diagram of shared variable states.




### Keeping track of who's set what

The result of □*svs* indicates which of you has set a value of which the other is still ignorant. It also shows which of you is aware of the shared variable's current value.

**The first pair: who has set a value that the other hasn't used?** One of the first two columns of the result of □*svs* contains a 1 when the latest value of the shared variable has been set by one partner, but not yet used by the other.

There's no way the first two columns could both be 1. However, they could both be 0: when both partners are aware of the shared variable's value, neither of them can have set a value of which the other is unaware, and so the first two elements are 0 0. The first two elements are also 0 0 when sharing has just been established, and neither partner has yet set the shared variable.

**The second pair: who's aware?** The second pair of columns in the result of □*svs* indicates which of the partners is aware of the shared variable's current value. There are three possibilities: only you know; only your partner knows; you both know.

The partner who last set a value for the shared variable must be aware of it (having just set it!). The other partner can be aware of the shared variable's value only if it has used the variable since it was set. Thus, the last two columns may contain either one 1 or two 1's.

When sharing has just been established, it is presumed that neither processor has yet set a value for the shared variable (but that each knew what value the variable had in their respective workspaces before it was shared.)

### Possible states of the shared variable

While a variable is actually shared, only three distinct states of □*svs* are possible. (The fourth state is all zeros, indicating that the name is not shared.)

POSSIBLE VALUES OF □*svs* AND THEIR INTERPRETATION

| Who has set a value that the other hasn't used? | | Who knows the current value? | | |
|---|---|---|---|---|
| You | Your partner | You | Your partner | Interpretation |
| 0 | 0 | 1 | 1 | You and your partner are both aware of the current value, and neither has since set the variable. (This is the initial value when sharing is first established). |
| 1 | 0 | 1 | 0 | You've set the shared variable, but your partner hasn't used it. |
| 0 | 1 | 0 | 1 | Your partner has set the shared variable, but you haven't used it. |
| 0 | 0 | 0 | 0 | The name isn't the name of a shared variable. |

### Access control

If you set your shared variable with a message for your partner at the same time that your partner is setting it with a message for you, at least one of those messages is lost. If you impatiently use the value of your shared variable before your partner has had time to put a reply there, you may be rereading your own earlier request, or your partner's reply to an earlier request. You want to avoid that kind of confusion.

Either of the partners may assure effective communication between them by setting the appropriate elements of the **access control vector**. Access control works by **inhibiting** you from doing something too soon. For example, it can inhibit you from setting a new value for the shared variable until your partner has referred to it. Or it can inhibit you from using the variable before your partner has set it for you.

When the access control is set to inhibit an action, it simply delays that action until the necessary conditions have been fulfilled. Consider the expression that was the core of the sample programs that use shared variables (for example, in chapter 5 or chapter 13):

```
r←shv ⊣ shv←cmd
```

Assuming that the access controls have been set properly, there's no risk of using the result too soon, nor of delaying longer than necessary.

Like its state vector, a shared variable's access control is a four-element boolean vector. By making the appropriate elements 1, either partner may specify that certain actions will be inhibited. You set the access control vector with the dyadic function ⎕*svc*. (Before either partner has set it, the access control vector is 0 0 0 0; you can't set a name's access control until you have offered to share it.)

The right argument of ⎕*svc* is the name of a shared variable, and the left argument is the four-element vector indicating which actions you propose to inhibit. Alternatively, the right argument may be a matrix of names, in which case the left argument is either a corresponding matrix of four-element vectors, or a single vector applying to all of them.

The significance of the four elements of the access control vector can be summarized as follows:

| Inhibit Re-set | | Inhibit Re-use | |
|---|---|---|---|
| ⎕ | ⎕ | ⎕ | ⎕ |
| **You** | **Your Partner** | **You** | **Your Partner** |

Wherever a variable's access control vector contains a 1, the corresponding action is inhibited. The effect of a 1 in each of the four positions is as follows:

1 You are inhibited from again setting the value of the shared variable until your partner knows what it is (because the partner has referred to it: that is, used it or set it).

2 Your partner is inhibited from again setting the value of the shared variable until you know what it is (because you've referred to it: that is, used it or set it).

3 You are inhibited from re-using the value of the shared variable until your partner has set it.

4 Your partner is inhibited from re-using the value of the shared variable until you have set it.

**Effective access control**

The **effective** access control is the control that's actually in effect (to distinguish it from the access control either one of the partners may have requested separately). The effective access control is set by uniting the access control vectors proposed by each of the partners separately. (The system keeps track of which partner requested what.) The effective access control vector has 1 set wherever **either** partner proposes it.

This rule means that either partner is free to set 1 in any of the four positions (that is, to inhibit any of the four possible actions). However, one partner cannot change to 0 of any element that the other has set to 1. You can always increase the level of inhibition for yourself or your partner, but you can remove only those inhibitions that you set alone.

The APs distributed with SHARP APL/PC all set access control to all 1s as soon as they can. However, that doesn't mean that you shouldn't do so also. When you've just offered to share a variable with an AP, you don't want your first instruction to be lost. Since you aren't certain when the AP will respond and will set its side of the access control, you shouldn't assume that access control has been taken care of: the AP may not yet have had time to do so when you're ready to send your first command. It's prudent **always** to set the access control to 1 1 1 1 yourself.

Figure 9-1: Shared variable transitions

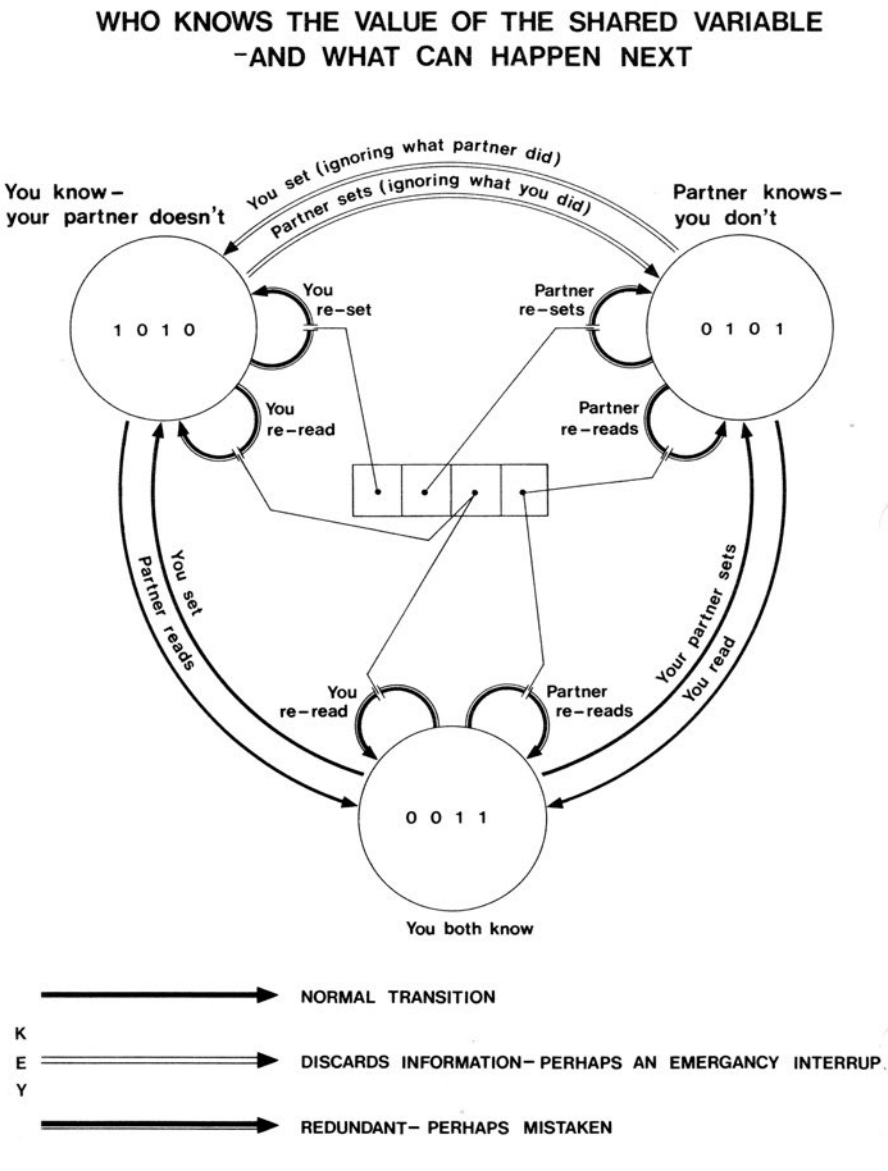


**Diagram summarizing shared variable communication**

The three circles represent the three states of the shared variable; the caption describes each state in English. Inside each circle you see the four-element vector by which that state is identified in the result of □*svs*.

(You may think that this diagram is making a big production out of a simple matter of communication. Indeed, when things are set up correctly, you won't need to be concerned with all this. But when you've done something wrong and are debugging a defective communication link, this diagram is handy.)

The curved paths show the various ways you could get from one state to another. For example, if you're in the state shown by the lower circle ("You both know"), and then you set the shared variable, the path labelled "You set" takes you to the state labelled "You know but your partner doesn't".

**Orderly communication.** Normal orderly communication is represented by the pairs of solid lines linking the lower state ("You both know") to one or the other of the states at the top. A normal sequence is: you start in the lower circle; then you set the shared variable, taking you to the upper left; then your partner uses it, bringing you back to the lower circle; then the partner sets it, taking you to the upper right; then you use it, bring you back to the lower circle again; and so on indefinitely.

The other paths represent various exceptional transitions. Mostly they arise from programming error, or emergencies. Avoiding the erroneous ones is what □*svc* is for.

**Representation of access control.** At the center of the diagram is shown the effective access control vector established by the way you and your partner have set □*svc*. For each element of the access control vector, a line shows which paths are blocked by a value of 1.

**Exceptional routes.** Since there are two of you, and either of you could set the shared variable or could use it, there are four arrows coming out from each circle. Some of the arrows don't go anywhere, but lead back to the same circle. That happens when either of you repeats an action--for example, if you already know the value of the shared variable but you use it again. These paths are shown by dotted lines. There are six of them altogether. At best, they are redundant. At worst, they may arise by mistake (for example, you **think** you're about to read new information, but in fact your partner hasn't sent any.) **All six of these redundant paths can be blocked by an appropriately placed 1 in the access control vector.**

There are two paths across the top shown by double lines. They represent interruptions: when one of you ignores what the other has just set, and instead sets a new value in the shared variable. This is something you don't want to do all the time, but which must be possible, as a way of signalling to a partner that you have to break out of the ordinary pattern of exchange. Neither of these

paths can be inhibited. That you send an interrupting message to your partner doesn't guarantee that the partner will notice it or heed it.

### Attempting to carry out an action that is inhibited

Suppose you have a variable whose state vector at present contains 1 0 1 0 ("You've set") and whose control vector has a 1 as its first element (to inhibit you from re-setting until your partner knows what it is). You attempt to give the shared variable a new value anyway. What happens? You experience a delay. There is no interrupt or error message, but the system waits to make your specification until your partner uses (or sets) the variable. The delay lasts indefinitely--as long as it takes.

If you decide you'd rather not wait, you can signal a **strong interrupt** by pressing <Break> (<Ctrl ScrollLock>) at least twice. (You may want to do this if you suspect that there's something wrong with the AP, or that your attempt to re-set the shared variable was ill conceived.) This aborts your attempt to use the shared variable. The system displays the error message *sv interrupt*, and displays the statement it was attempting to execute.

A similar delay occurs when a variable's state vector has a 1 in the third position (meaning that you know what the variable's value is), and there's a 1 in the third element of the variable's access control (to inhibit you from rereading until after your partner has set). There's an indefinite delay until your partner has again set a value for the shared variable. As before, you can interrupt the attempt to use the interlocked variable by signalling a strong interrupt.

**Watch out:** While you're trying to discover what went wrong with a program interrupted by an error, it may seem natural to attempt to display a shared variable, just to take a look at what's going on. But such a display counts as a reference to the variable, and might be inhibited.

If you wish to be able to make repeated use of the value of an interlocked shared variable, one technique is to immediately assign its value to another (unshared) variable, so you can refer to it as often as need be. Another technique (which you'll see if you examine some of the defined functions in the distributed workspace *fscr*) is to use 6 ⎕*ws* '*shv*' to examine the value of the shared variable *shv*.

⎕*pack*, 4 ⎕*ws*, **and** 6 ⎕*ws* **applied to a shared variable.** When you use the name of a shared variable in the argument of any of these three functions, as far as the Shared Variable Processor is concerned, this does **not** constitute a reference to the shared variable. With monadic ⎕*pack* and 6 ⎕*ws* you get the last value visible in your workspace, and with 4 ⎕*ws* you get the size of the last value visible in your workspace. Thus, you only get a value that you already know (either because you've already made some

other use of it, or because you're the one who set it). Using any of these three functions has no effect on the shared variable state.

### Calculating in advance which actions will not be delayed

By examining a variable's state vector and its access control vector, you can tell which actions will **not** be subject to delay. Recall the four possible actions: you set, your partner sets, you use, and your partner uses. For a variable α, an action you're free to take immediately (that is, with no delay) is any of those indicated by a 1 in the expression

```
(⎕svs 'α') ⍱ ⎕svc 'α'
```

### Meaning of some possible settings of a variable's access control vector

| | |
|---|---|
| 0 0 0 0 | No interlock. Such a setting is reasonable only when you know that no other constraints will prevent the two processors from getting out of step. This might happen if the programs that use this shared variable without interlocks also make use of another variable that is interlocked; that technique is used by the two-variable method of using FSCR, decribed in chapter 13. |
| 1 1 0 0 | Neither of you can set a new value until the other has referred to it. (Sometimes called half duplex.) |
| 0 0 1 1 | Neither of you can reread the shared variable until the other has set it. (Sometimes called half duplex.) |
| 1 0 0 1 | You can't re-set until your partner has referred to it, and your partner can't re-use it until you've set. (Sometimes called simplex transmission from you to your partner; appropriate if you are having a rather one-way sort of conversation, as you might while sending instructions to a printer or plotter.) |
| 0 1 1 0 | You can't reread until your partner has set, and your partner can't set until you've referred to it. (Sometimes called simplex transmission from your partner to you.) |
| 1 1 1 1 | Full interlock: reversing half duplex. |

### State-change signal

Suppose you just want to wait until an AP accepts an offer. How can you say "wake me when it answers"?

Or suppose you have a function that uses several shared variables (for example, using two serial ports to communicate with two different external systems). The function may, on occasion, reach a point where it has nothing to do immediately, but is waiting for a

reply to come in from any of its shared variables--but it can't tell which one will be set next. How can you can have the function wait until something happens that may require its attention?

You wait for a change in the state of your shared variables by using the system variable ⎕*sc*.

⎕*sc* is a variable automatically shared with the system. Whenever you use ⎕*sc*, you get the value that the system has set. That value is either 0 or 1. The interesting property of ⎕*sc* is usually not **what** its value is, but **when** you get it. While you are using shared variables, between any two successive uses of ⎕*sc*, the system must have detected a **state change**. If there hasn't been a state change, execution is delayed until there is one. So when you receive your value of ⎕*sc*, you're assured that something other than yourself has done something that affects the shared-variable situation.

**What counts as a state change?** A state change is an action (or several actions) outside your workspace but affecting your use of shared variables. "State change" is thus a rather general category. It includes any new offer to share extended to you; any set of a variable you are already sharing, or any set of its shared-variable control; any change in the degree of coupling of a variable you have already offered; or the first use by a partner of a value that you set.

Once your active workspace has established a valid clone ID, each time a state change occurs, the system sets the value of ⎕*sc* to 1. (⎕*sc* has the value 0 only when you have no valid clone ID.) You cannot use that value more than once; when you next use ⎕*sc*, the value you get will be a value that the system set **after** it set later than the last time you ued it, indicating that at least one state change has occured since you last used ⎕*sc*. Thus the value of ⎕*sc*--when you succeed in using it--is always 1.

If a state change has occured since you started sharing or since the last time you referred to ⎕*sc*, you get that value 1. But if there hasn't been a state change, the system doesn't return the value of ⎕*sc* **until there's been a state change**. Hence, using ⎕*sc* amounts to waiting indefinitely until some state change takes place. However, if a potential partner retracts an offer that was made to you but which you have **not** accepted, that doesn't count as a state change.

A reasonable sequence for using ⎕*sc* is:

1. Using ⎕*svs*, verify that there's nothing you can do at present with the relevant shared variables.

2. Use ⎕*sc* in an expression such as

```
→(⎕sc/label),error
```

Provided you have a valid clone ID, that will delay the branch to *label* until some state change occurs. But if you lack a valid clone ID (and you can't possibly be making effective use of shared variables without one), control passes immediately to *error*.

3. When you've received a 1 from □*sc*, use □*svs* or □*svq* to review the situation and decide what action is possible or appropriate.

### Saving a workspace containing shared variables

When a workspace is saved, the APL system requires that the value of each of its variables must be physically present within it. When you give the command )*save* or )*continue*, it's possible that you are sharing a variable whose value you have not yet used (your partner knows the value, but you don't). If you were to use the variable, the SVP would supply it. But when you save the workspace, the value must actually be physically present in the workspace (not merely available from the SVP if needed). So, before it saves the workspace, the system moves into the workspace the current value of each of the shared variables to which you haven't yet referred.

If it proves impossible to transfer the value of a shared variable because of insufficient space, the system displays the error message *ws full*, and does **not** save the workspace.

# Chapter 10: USER-WRITTEN AUXILIARY PROCESSORS

This chapter discusses how to write an auxiliary processor (AP).

The material is quite technical and assumes you:

- have studied chapter 9,
- are familiar with the operation of the distributed APs,
- understand the 8088 architecture and instruction set, and
- can read and write assembler programs.

An auxiliary processor is a program which is a part of the SHARP APL system. The SVP tries to protect the system from problems in the AP, but an incorrectly functioning AP can cause arbitrary damage to the system. An AP must be carefully written and debugged, and must follow the rules and guidelines laid out in this chapter.

Distributed file MUSIC.ASM contains assembler source for a simple AP that demonstrates most of the facilities available to an AP writer. After reading this chapter you should study the MUSIC AP carefully.

MUSIC.ASM, with the parts concerned with playing music discarded, can form the framework in which you can write your own AP.

## The SVP

The SVP is a program which is invoked via a software interrupt. It can access both its own storage and that of the program which issued the interrupt.

Programs written in compiled languages (such as C) can use the SVP by calling appropriate assembler-language routines.

Source statements for building an interface to the SVP are defined in distributed files APDEFNA.PCC (for programs written in assembler) and APDEFN.H (for programs written in C). If you wish to use C, you will have to determine the applicability of APDEFN.H for your C compiler.

This chapter, in conjunction with APDEFNA and MUSIC.ASM, provides enough information for you to write your own AP.

## THE TASK DISPATCHER

DOS does not support concurrent tasks, so the SVP must provide its own task dispatcher. This lets multiple tasks (APs, the SHARP APL File System, and APL) run concurrently, taking turns at obtaining processor cycles and operating system services.

The task dispatcher supports three types of tasks:

1. Interrupt tasks: These tasks are entered via hardware interrupts. DOS functions and most BIOS functions are not allowed in an interrupt task. They may issue an INT INTPOST instruction to request the scheduling of a normal task.

2. Normal tasks: An AP is a normal task. The task dispatcher runs a normal task by calling it as a closed subroutine with a CALL FAR.

Once dispatched, a normal task runs until it returns to the task dispatcher with a RET FAR. While dispatched, it may perform any DOS or BIOS functions.

Once control is returned to the task dispatcher, the task will not be dispatched again until posted by some other task, and then only after all other normal tasks have had a chance to execute.

3. Low-priority task: APL is the low-priority task. It runs only when no other task wishes to run. The low-priority task makes periodic entries to the task dispatcher, and when the task dispatcher has no normal task ready to run, it returns to the low-priority task.

### Task posting

A task may post itself or any other task. Posting self, then returning to the task dispatcher, allows all normal tasks, but not the low-priority task, to run before the task is dispatched again.

Each task is assigned a unique number called the Perproc Index (PPI) when it signs onto the SVP. This number is typically used by an interrupt task to post its related normal task.

When a task is dispatched, the post codes for the task are moved into the PCVPOST field of the task's PERPCV. The post codes are defined in APDEFNA.

The task must examine these post bits to determine why it was posted. More than one post may occur before the task is dispatched, so more than one post bit may be set.

Four post codes have been reserved for the AP writer. These user post codes may be specified as an argument to an INTPOST software interrupt and typically are used by an interrupt task to post its associated normal task.

A task posts a normal task (including itself) by issuing an INTPOST software interrupt with AX set to the desired post codes and BX set to the Perproc Index (PPI) of the task to be posted. A task's PPI is returned in the PERCMD PCP2 field as a result of a successful SVPON function.

## SVP STRUCTURES AND TERMINOLOGY

An AP uses four types of structures (control blocks):

PERPCV
PERCMD
PERSCV
ASDATA

These structures are defined in APDEFNA.

An **offer** is a variable with a degree of coupling of 1 (that is, a processor has offered it, but no processor has matched the offer).

A **share** is a variable with a degree of coupling of 2 (that is, the variable is currently shared by two processors).

### PERPCV: per processor control vector

Each AP must establish and maintain a PERPCV. The PERPCV identifies the AP to the SVP and to the task dispatcher. It also contains the values used by the task dispatcher to establish register contents when dispatching a task. Correct establishment of these values in the PERPCV before SVP signon is critical to the operation of an AP.

The PERPCV fields are:

PCVID
The AP's processor ID (an integer between 1 and 1000).

PCVNID
The AP's clone ID (a nonnegative integer, usually 0).

PCVPOST
This field is set by the task dispatcher each time the task is dispatched. The 16 bits indicate why the task was dispatched (retraction, new offer, system shutting down, etc). More than one bit may be set at a time.

PCVTCS
The IP:CS values to be set when the task is dispatched, stored in a form acceptable to the LES instruction.

PCVTSS
The SP:SS values to be set when the task is dispatched, stored in a form acceptable to the LES instruction.

PCVTDS
The DX:DS values to be set when the task is dispatched, stored in a form acceptable to the LES instruction.

PCVTBP
The BP register value to be set when the task is dispatched, in a form suitable for the MOV BP,PCVTBP instruction.

**PERCMD: per SVP command**

Each call of the SVP requires a PERCMD. The PERCMD fields are:

PCCMD
indicates the SVP operation to be performed.

PCFLAGS
must be 0.

PCP1, PCP2, PCP3
are three fields that hold arguments and results for an SVP operation. The contents depend on the SVP operation requested in PCCMD.

**PERSCV: per share control vector**

A PERSCV is used in SVPSVO and SVPSVQ operations. The PERSCV fields are:

SCVCOUNT
If an initial value is being passed to the SVP in an SVPSVO operation, SCVCOUNT contains the byte count of the value.

SCVSOS
This field is used to alter the treatment of an SVPSVR operation by either partner.

If both partners set SCVSOS to 0, a retract by either side may be thought of as putting the partner on hold: a later SVPSVO will recouple to the partner on hold.

If either partner sets SCVSOS to 1, the SVPSVR acts like "calling/called party disconnect": the offer will never be rematched. Setting SCVSOS to 1 is typically used when retraction is a signal to terminate the current communication with the AP. For example, if a retract from a file-writing AP indicates that the file should be closed, an SCVSOS of 1 avoids a race condition where the partner retracts (indicating it is finished with that file) and reoffers (intending to work with a new file) before the AP is dispatched and acts on the retraction.

SCVID
The processor ID of the AP (must match the value in PCVID).

SCVNID
The clone ID of the AP (must match the value in PCVNID).

SCVALUE
If an initial value is being passed to the SVP in an SVPSVO operation, SCVALUE contains the mainstore address of the value. Set SCVALUE to ¯1 if there is no initial value.

SCVCHRON
A positive integer used by the SVP to maintain a chronological order for shared variable offers (degree of coupling is 1). An AP may use the chronology to exclude the consideration of offers earlier than specified. The SVP resolves the ambiguity of two or more acceptable offers by choosing the oldest, that is, the one with the smallest chronology. SCVCHRON is usually initialized as 0.

SCVPO
This is set nonzero by a successful SVPSVO or SVPSVQ operation. It should be set to 0 before an SVPSVO, unless the PERSCV was filled in by a SVPSVQ. The SCVPO is used as an abbreviated name of the shared variable in other SVP operations (similar to the "file handles" used in DOS). The nonzero contents of SCVPO, when used as an SVP parameter, is called a PO (per offer).

SCVOTHER
The processor ID of the partner. If SCVOTHER is 0, then SCVNOTHR must also be 0. An SCVOTHER of 0 is a general offer, which may be matched by any specific offer to this processor.

SCVNOTHR
The clone ID of the partner.

SCVNAMEL
The length of the shared variable name, or the length of the longest acceptable name. The SVP has an 11-character limit on the length of names.

SCVNAME
The name of the shared variable, if a specific name is desired. If a match with any name is desired, the first byte should be set to hexadecimal 0. A successful SVPSVO or SVPSVQ will fill in the name. If sharing with APL, the names must be in APL symbols, not ASCII (see APL character equates *qa* through *qau* in APDEFNA). If sharing with another AP, the names are arbitrary character strings (except that the first character of the name must be nonzero).

## ASDATA: APL SVP data

ASDATA defines the format of data passed between an AP and APL. The values are self-describing: they contain the type (character, boolean, integer, floating-point, and complex), the rank, the shape, and the elements. See the structure ASDATA in APDEFNA for a complete definition.

The SVP does not validate any data. However, APL does validate the data it receives from the SVP and any value that is not valid according to the ASDATA definition is rejected and is reported to the APL user as a *value error*.

## SVP OPERATIONS

An SVP operation is performed by executing software interrupt INTSVP (interrupt 61H) with BX,AX as the address of a PERCMD. The PERCMD indicates which operation to perform in field PCCMD and contains parameters in fields PCP1, PCP2, and PCP3.

SVP operations need memory addresses for a PERCMD and possibly for a PERSCV or an ASDATA. An SVP memory address will be called a **mainstore address**.

A mainstore address is a four-byte integer, and is stored in memory with the bytes in reversed order (because of the machine architecture). For example, address 68364, which is 00010B0CH, is stored in memory as 0C0B0100.

The mainstore address of a PERCMD is loaded into BX,AX with the high-order bytes in BX and the low-order bytes in AX. For example, if 68364 was the mainstore address of a PERCMD, then BX would contain 0001H and AX would contain 0B0CH.

A mainstore address is not a segment/offset address. The procedure SEGTOMS in MUSIC.ASM converts a segment/offset address into a mainstore address.

The SVP services the request as a closed subroutine, and returns control to the requester with a return code in AX. The codes returned in AX are documented in the section **Return codes**. The SVP will have modified the PCP1, PCP2, and PCP3 fields of the PERCMD, as well as a PERSCV or ASDATA addressed by these fields as is appropriate for the operation.

## SVP COMMANDS, PARAMETERS, AND RESULTS

Each SVP operation has a command name. These names have a prefix of SVP and are defined in APDEFNA. In many cases there is a direct analogy to an APL system function. For example, SVPSVO corresponds to *□svo*, and SVPSVR corresponds to *□svr*. In some cases the analogy is less direct. For example, an APL set of a shared variable corresponds to an AP's doing an SVPSIS (set initial selection) followed by an SVPSDT (set data transfer).

The SVP operation to be performed is indicated by storing the appropriate command (for example, SVPSDT) in the PERCMD PCCMD field.

An SVP operation takes 0, 1, 2, or 3 parameters. These parameters are passed in the PERCMD PCP1, PCP2, and PCP3 fields. A parameter can be a four-byte integer (for example, a PO or a LENGTH); or it can be the mainstore address of a PERSCV or an ASDATA.

Each SVP operation is defined in this section with a list of the PERCMD parameters. Parameters indicated as PERSCV and ASDATA are mainstore addresses. Other parameters are values that are stored directly in the PERCMD field. For example,

```
SVPSDT PO,ASDATA,LENGTH
```

indicates PCCMD must be set as SVPSDT, PCP1 must be set as the PO of the variable, PCP2 must be set as the mainstore address of the ASDATA containing the data to set in the variable, and PCP3 must be set as the length of the data in ASDATA.

When the PERCMD is set as required and BX,AX are set to the mainstore address of the PERCMD, the SVP is called by executing an INT INTSVP.

The return code of the operation is returned in AX. Return codes are defined in APDEFNA with prefixes of SVZ, and are documented in the section **Return codes** below.

Any operation can have a return code of SVZNE, SVZNA, SVZARG, SVZSPE, or SVZNSO, with or without the SVZNOS modifier. The following sections state what other return codes, if any, are possible for a particular operation.

Other results are returned in PERCMD fields PCP1, PCP2, and PCP3, and in a PERPCV or ASDATA addressed by a PERCMD field, as appropriate.

### Signing on: SVPON PERPCV

The first SVP command performed by an AP must be to sign on to the SVP. This identifies the processor to the SVP and task dispatcher.

The PERCMD PCCMD field contains SVPON and the PCP1 field contains the mainstore address of a PERPCV initialized as described in the PERPCV description.

SVPON return codes: SVZNIU, SVZASO, SVZPPF.

If the signon was successful, PCP2 contains the Perproc Index (PPI) assigned to the AP. This is the value used by an interrupt task to post the AP.

**Scanning for offers: SVPSVQ PERSCV**

An AP can determine whether any variables which meet certain criteria have been offered to it by scanning for offers.

The PERCMD PCCMD field contains SVPSVQ, and the PCP1 field contains the mainstore address of a PERSCV initialized as described in the PERSCV description.

SVPSVQ return codes: SVZNOF.

If the SVPSVQ was successful, the PERSCV has been modified as follows:

SCVCHRON
The chronology of the located offer.

SCVPO
The SVP internal name for the variable. This value is used to identify the variable in all subsequent operations and is referred to as the PO (per offer).

SCVOTHER,SCVNOTHR
Processor ID and clone ID of the offering AP.

SCVNAMEL,SCVNAME
Length and name of the offered variable. If SVPSVQ was invoked with the first character of SCVNAME nonzero, these fields are unchanged. If the first character was zero, SCVNAMEL has been changed to the length of the name which was found, and SCVNAME contains the name unless the name is longer than the value of SCVNAMEL when SVPSVQ was invoked, in which case the name has been truncated to this limit.

**Offering to share: SVPSVO PERSCV**

An AP can offer a variable to another AP or to APL. This can be a new offer, or it can be an offer to match an outstanding offer, resulting in a share.

Except in the cases where SCVNAME has been truncated or SCVALUE has not been properly initialized, the PERSCV modified by an SVPSVQ is a proper argument for SVPSVO and may be used to match the offer. The resultant degree of coupling must be

examined, however, in case the offering processor has retracted the offer between the SVPSVQ and the subsequent SVPSVO.

If the PERSCV is not the result of a successful SVPSVQ, it must be initialized as described in the PERSCV description.

SVPSVO return codes: SVZPSF.

If the SVPSVO was successful, PCP2 contains the degree of coupling of the variable. Zero indicates that a PERSCV which specified a nonzero SCVPO did not match the variable which has the specified internal name; the PERSCV has not been altered. One indicates an offer was extended, and two indicates that an offer was accepted and the variable is shared.

An SVPSVO executed with a PERSCV which describes a variable which has already been offered is an inquiry for the degree of coupling.

SCVNO contains the chronology of an offer which was extended or accepted.

SCVOTHER and SCVONID will contain the identification of the processor with which the variable is shared (the degree of coupling is 2).

### Retracting an offer: SVPSVR PO

The PERCMD PCCMD field contains SVPSVR, and the PCP1 field contains the PO of the offer to retract. The PO is the internal name of the variable.

If the SVPSVR was successful, PCP2 contains the degree of coupling before retraction.

If the degree of coupling was 1, the offer has been erased and the shared variable no longer exists. If 2, the shared variable remains as an offer to the retracting AP.

### Access control

Associated with each shared variable is a four-element access state vector and two four-element access control vectors. The effect of these is described in chapter 9. The AP's contribution to the access control can be set, and both the access state vector and the combined access control vector can be examined.

### Setting access control: SVPSVC PO,CV

The PERCMD PCCMD field contains SVPSVC, PCP1 contains PO (internal variable name), and PCP2 contains CV, which is a value between 0 and 15 inclusive. CV is the four bits to OR with the access control vector.

If the SVPSVC was successful, PCP2 contains the current access state vector in the first four bits of the low-order byte, and contains the access control vector determined by OR-ing the value just set with the partner's contribution, in the last four bits.

### Reading the access state and access control: SVPSVS PO

This is the same as an SVPSVC operation, but no CV parameter is provided and so no change is made to the access control.

### "Use" and "set"

A shared variable is used (read) or set (written) by a sequence of two operations. The first is an **initial selection** which locks the variable, that is, obtains control of the variable so that the partner is excluded from initial selection operations, and which also returns information about the current status of the variable so that scheduling decisions, storage allocation, etc. can be performed while the access state remains unchanged. Operations which transfer data are only accepted when the variable is under control obtained by an appropriate initial selection.

### Initial selection operations

| | |
|---|---|
| SVPUIS PO | Use initial selection (ready for read) |
| SVPSIS PO | Set initial selection (ready for set) |
| SVPCIS PO | Copy initial selection (ready for copy) |

The PERCMD PCCMD field contains the appropriate command (SVPUIS, SVPSIS, or SVPCIS) and PCP1 contains the PO of the variable to be locked.

Return codes: SVZLOCK, SVZIVS, SVZNV.

If the initial selection was successful, PCP2 contains the length in bytes of the current shared variable value, or contains ¯1 to indicate there is no value present in SVP storage.

### Data transfer operations

| | |
|---|---|
| SVPUDT PO,ASDATA,LENGTH | Use data transfer (read data) |
| SVPSDT PO,ASDATA,LENGTH | Set data transfer (set data) |
| SVPCDT PO,ASDATA,LENGTH | Copy data transfer (copy data) |

The PERCMD PCCMD contains the appropriate command (SVPUDT, SVPSDT, or SVPCDT), PCP1 contains the PO of the variable, PCP2 contains the mainstore address of the value, and PCP3 contains the length of the value in bytes.

SVPUDT return codes: SVZIVS.

SVPSDT return codes: SVZSVSF, SVZIVS, SVZVTL.

SVPCDT return codes: SVZIVS.

A successful SVPUDT moves the data from SVP storage into the ASDATA in the AP, updates the access state, and releases control.

A successful SVPSDT moves the data from the ASDATA into the SVP storage, updates the access state, and releases control.

A successful SVPCDT copies the data from SVP storage into the ASDATA in the AP. SVPCDT is intended as an intermediate step in sequences such as are required for APL indexed specification. It does not release control and does not change the access state. It may follow any initial selection operation which obtains control.

**Release control: SVPNDT PO**

An SVPNDT operation releases control of a variable when control has been obtained with an initial selection, but when the data transfer is not desired.

The PERCMD PCCMD field contains SVPNDT and PCP1 contains the PO of the variable to release.

## RETURN CODES

The value in AX at the completion of an SVP request is an integer return code.

The parity of the return code is used to indicate the degree of coupling at the completion of any command on a specific shared variable. If the degree of coupling is 1, then SVZNOS (which is 1) is added to the return code, making it odd. An even return code indicates a degree of coupling of 2.

A normal-end return, SVZNE or SVZNE+SVZNOS, indicates that the request was accepted and executed successfully. The results contained in the PERCMD fields PCP1, PCP2 and PCP3, and a PERSCV or ASDATA may be examined for further information.

The following return codes and mnemonics are defined in APDEFNA:

**0** (SVZLOCK)
An operation was rejected because the access controls prevent access.

**1** (SVZNOS, i.e. no share)
This return code is added to other return codes if the variable has a degree of coupling of 1. For example, SVPSVO returns SVZNE if the share matched an existing offer, resulting in a degree of coupling of 2; or returns SVZNE+SVZNOS if there was no matching offer.

**2** (SVZSVSF, i.e. shared variable storage failure)
An SVPSDT operation was rejected because of a temporary shortage of shared variable storage. The AP will be posted when shared variable storage is released so that it may retry the operation. This is a transient condition, but may indicate that you should start the SVP with more storage (the B start parameter; see chapter 2 **Getting started**).

**4** (SVZILI, i.e. incorrect length indicator)
The byte count specified by an SVPUDT or SVPCDT does not match the count returned to the user by a previous SVPUIS or SVPCIS. The control over the variable obtained by the initial selection is released. No data transfer occurs.

**6** (SVZNOF, i.e. no offer found)
An SVPSVQ found no offer. There are no offers which match the specifications contained in the PERSCV; the PERSCV is unchanged.

**8** (SVZNE, i.e. normal end)
Normal completion.

**10** (SVZNIU, i.e. number in use)
An SVPON found the processor ID and clone ID are already in use.

12 (SVZNSO, i.e. not signed on)
Returned for all requests except signon if the indicated identification is not signed on.

14 (SVZASO, i.e. already signed on)
An SVPON found the processor is already signed on.

16 (SVZIVS, i.e. invalid sequence)
An SVPUDT, SVPSDT, or SVPCDT was not preceeded by an appropriate and successful initial selection.

18 (SVZNV, i.e. no value)
An SVPUIS or SVPCIS found no value available in SVP store.

20 (SVZSPE, i.e. storage protection exception)
A mainstore address pointed either into the first 1024 bytes of memory, or past the end of memory.

22 (SVZNA, i.e. not available)
The SVP is not running.

24 (SVZPSF, i.e. PERSHARE failure)
An SVPSVO requires a PERSHARE and all of them are in use. Either retract a variable and try again, or restart the SVP with more PERSHAREs (the P start parameter; see chapter 2 **Getting started**).

26 (SVZPPF, i.e. PERPROC failure)
An SVPON requires a PERPROC, and all of them are in use. Restart the SVP with more PERPROCs (the P start parameter; see chapter 2 **Getting started**).

28 (SVZVTL, i.e. variable too large)
An SVPSDT specified a value that is too large to fit in SVP store. Either set smaller values, or restart the SVP with more SVP store (the B start parameter).

30 (SVZARG, i.e. argument error)
An argument has failed a validity check. For example, the PO specified was not for a variable offered by this processor.

## TERMINATE BUT STAY RESIDENT

The AP is initiated as a DOS task, probably by a line in a BAT file. After the AP has completed its initialization and signed on to the SVP it must return control to DOS so that the remaining APs and finally CP8X can be started.

When the AP is started by DOS, it has all the remaining memory allocated to it. When it returns control to DOS it must indicate what memory remains allocated to it, and what memory is released for use by subsequent APs, CP8X, APL, and the workspace.

An AP returns control to DOS with software interrupt "terminate but stay resident" (interrupt 27H), indicating the amount of memory for DOS to leave allocated to the AP, and the start of memory available for the next program.

## TASK TERMINATION

When you terminate APL (for example, with *)off*), you automatically terminate all APs, SVP, and the File System. An orderly termination--one that leaves the PC in a useful state--requires cooperation from all APs. In particular:

- files must be closed,
- interrupt vectors must be returned to their previous states, and
- I/O devices must be disabled.

An AP's failure to terminate properly can result in a machine state that requires a power-off/on reset.

When APL is terminated, it will retract its shared variables and your AP will have been posted to run in order to process these retractions.

Eventually, the task dispatcher will post your AP with the PERPSHUT post code. The PERPSHUT post code may be accompanied by other post codes. If your AP shared variables with other APs, it may be posted with PERPSHUT more than once.

The PERPSHUT post code indicates a system shutdown. When an AP is dispatched with a PERPSHUT post code, the AP must tidy up as required and return control to the task dispatcher in the normal manner.

The task dispatcher, not the APs, deallocates storage.

APs must not terminate other than as described here. In particular, an AP must never execute the DOS TERMINATE function once it has signed on to the SVP.

## THE MUSIC AP

Probably the best way to learn how to write your own AP is to study MUSIC, the sample AP that is distributed with the system.

The assembler source for the MUSIC AP is in distributed file MUSIC.ASM. The MUSIC AP is also distributed ready to run as MUSIC.EXE.

MUSIC plays notes on the PC's speaker. It uses the PC's 18.2-Hz timer interrupt to control the duration of notes, and gets information about the frequency and duration of notes from a shared variable.

The MUSIC AP uses processor identification 440.

You should get an assembly listing of MUSIC to read along with the descriptive material in the sections below.

You should probably also link MUSIC and run it as an AP. Improving and modifying MUSIC is a good way to gain experience before writing your own AP.

### The MUSIC argument

The variable shared with MUSIC must be set with a three-column character matrix. Each row specifies the frequency and duration of the note. The first two characters are used as the argument to an OUT instruction to get the PC speaker frequency divider. The note is played for the number of 18.2-Hz ticks specified in the third character.

The following expression converts frequencies expressed in hertz to the characters required by the MUSIC argument:

```
⍉⎕av[256 256 ⊤⌊0.5+595238÷1⌈⍵]
```

### MUSIC initialization--MUSICEP

MUSIC starts execution at the procedure MUSICEP as a DOS task. It establishes segment registers, and sets up a PERPCV containing the register values it wishes to run with when it is dispatched by the task dispatcher. It also establishes mainstore addresses (four-byte integers, not segment/offsets) of the PERCMD, the PERSCV, and a buffer for the ASDATA.

It then checks to see if the SVP is running (by looking for a handler for software interrupt INTSVP), and terminates with a message if the SVP is not running. It then calls GOSVP to issue an SVP request for signon. If the signon fails, it terminates.

It then obtains control of the 18.2-Hz timer interrupt in a nondestructive, race-free fashion. This is done in procedure GRABTIMER, with interrupts disabled, by copying the existing

timer interrupt handler address to a safe place and placing its own handler address, MYTICKINT, into the interrupt address. Procedure MYTICKINT is now an interrupt task and sees all timer interrupts. It will perform any required function, then pass the interrupt on to the next handler. This technique allows a chain of interrupt handlers to be constructed.

MUSIC then issues a terminate-but-stay-resident request (INT 27H), returning control to DOS, but ready to be run by the task dispatcher.

MUSIC is now essentially idle, and aside from the MYTICKINT interrupt handler code being executed at 18.2 Hz, the AP will not be dispatched until another processor causes it to be posted, or the system shuts down.

### MUSIC interrupt task--MYTICKINT

MYTICKINT preserves its caller's registers (since the interrupt may occur at any time), establishes its segment registers, and then checks to see if it has work to do by examining bits in TICKWORK.

The routine does two kinds of work: enabling the speaker, and decrementing the time counter. If TICKENAB is set, procedure PLAY has obtained a new value from the SVP, and is waiting for a clock tick to start playing. It turns off TICKENAB, switches to playing mode by turning on TICKDEC, and enables the speaker. If TICKDEC is set, the timer code merely decrements NOTETIME and exits to the next interrupt handler if NOTETIME is nonzero.

When NOTETIME has decremented to zero, the current note is complete, and it is time to get the next one. It checks to see if another note exists, and if not, disables the speaker, sets TICKWORK to indicate no further work, issues an INTPOST for the MUSIC normal task, and exits to the next interrupt handler. If another note does exist, it sets the frequency into the speaker frequency divider, sets NOTETIME to the duration of the note in 18.2-Hz ticks, and exits to the next interrupt handler.

### MUSIC normal task--PLAY

The MUSIC task called by the task dispatcher is the procedure PLAY. It examines the PCVPOST codes for work. When it finds something to do, it calls the appropriate function, and then continues the search of PCVPOST codes. The continued testing is mandatory, since more than one post code may appear at the same time. Note in particular the test for PERPSHUT which calls procedure GOSHUT, which turns off the speaker, returns the timer interrupt to its previous value, and returns to the task dispatcher.

When an offer is posted (PERPOFFR), the procedure GOOFFER is executed. It checks to see if a variable is already shared. If so, it ignores the offer, since it only shares one variable. An AP may share any number of variables but MUSIC only shares one. MUSIC

issues an SVPSVQ operation to search for the offer. The scan is quite simple, since the AP is willing to accept any variable name. If the scan fails (perhaps because the offer was quickly withdrawn), GOOFFER returns. Otherwise, the scan operation has filled in appropriate fields of MYSCV with enough information to allow it to be used with SVPSVO. The share is performed, and if it works, a check for an initial value is performed by calling GOSET. If the share failed, a retract operation is performed, so that the AP does not leave an outstanding offer which may never be matched.

When APL sets the variable, the code at GOSET is executed. This performs an SVPUIS to obtain control of the variable, and to find out the size of the variable. This prevents the partner from changing the variable between the time MUSIC determines its size, and the time when MUSIC requests its value. If SVPUIS fails, GOSET returns after calling the BEEP procedure which sets the variable to "*bad*". Otherwise, MUSIC issues an SVPUDT to move the value of the variable to SVDAT, after ensuring that the variable is not too large to fit. If the value is not a three-column character matrix, a value of "*bad*" is set.

If the task was posted by TICKINT, then the current musical phrase has completed playing, and a value of "*ok*" is set. This allows a reversing half-duplex protocol to be established by the APL user. Try multiple sets with `1 ⎕svc 'x'` or `0 ⎕svc 'x'`.

If the value is acceptable, GOSET turns off the speaker, sets NOTEINDEX and NOTELIM as the index of the first and 1+last element of the variable, and sets NOTETIME and TICKENAB to start playing the first note at the next timer tick. This ensures that the length of the first note is correct, by making it start on a timer tick.

When a retract occurs, the AP retracts the variable on its side as well, so that another offer may be matched later.

### MUSIC termination--GOSHUT

Terminating MUSIC requires some care to ensure that the system will continue to function properly. When APL terminates, it will retract its shared variables and the task dispatcher will eventually post MUSIC with a PERPSHUT post code. MUSIC then returns the timer interrupt vector to its previous state (procedure UNGRABTIMER). Once MUSIC returns control to the task dispatcher after the shutdown post (PERPSHUT) is issued, it will not be dispatched again.

# Chapter 11: DOS INTERFACE AUXILIARY PROCESSOR (DOSX)

This chapter describes the DOS Interface Auxiliary Processor (DOSX). DOSX provides access to DOS files, parallel printers, some DOS function calls, memory (peek and poke), and user-written machine-language programs.

This chapter assumes you are familiar with DOS facilities and terminology. If not, you should refer to the IBM PC publications **Disk Operating System** and **Disk Operating System User's Guide**.

Actively experiment with the system while reading the material. A few experiments are worth a thousand words.

## The shared variable

A single shared variable is used to communicate with DOSX. The variable must be a character vector. The user sets the variable with an argument consisting of a command and appropriate parameters. DOSX responds by setting the variable with a result consisting of a return code, a subcode, and any result data.

The variable can be any valid name of up to 11 characters. DOSX sets access control so that the variable is fully interlocked.

You can share more than one variable with DOSX. For example, if two variables are shared, one can be used to read data from one file and the other to write data to another file. Each variable shared with DOSX is independent of the others.

## The argument

command [p1[,p2[,p3]]][;data]

The argument to DOSX must begin with a valid command name, which may be preceded by blanks. Commands cannot be abbreviated (for example, *cd* is an invalid command, not an abbreviation of *chdir*).

Some commands take parameters, which must be separated from the command by at least one blank. Parameters are positional and are separated by commas; blanks are not allowed in parameters.

Some commands take variable-length data as a parameter which runs from the semicolon to the end of the variable. This data can include any character, including blanks.

## The result

rc sc;[data]

The result from DOSX is a character vector that starts with a return code, subcode, and semicolon. The return code and subcode are two-digit numbers separated by a blank. If data is returned, it follows the semicolon.

An **rc** of 00 indicates the command completed successfully. Usually an **rc** of 00 has an **sc** of 00. The statement `'00'≡2↑result` is a test for successful completion. The definitions of **rc** and **sc** values are given in the table at the end of this chapter.

### DOSX start parameters

**Bn**
specifies the length of the largest character vector that can be set in a variable shared with DOSX. **n** may be any integer from 200 to 32767. If you omit B, the largest settable character vector may be 2000 characters long.

**Qn**
indicates whether DOSX is to display its start message. **n** may be either 0 (which specifies that the message is not to be displayed) or 1. If you omit Q, the message is displayed.

**Un**
specifies the processor number to be used by DOSX. **n** may be any integer from 2 to 999. If you omit U, DOSX uses processor number 710.

**Sn**
specifies the number of variables that can be shared at once with DOSX. **n** may be any integer from 1 to 255. If you omit S, up to 4 variables can be shared at once with DOSX.

**Wn**
specifies the printer width (see **Printer support** for more information about this parameter). **n** may be any integer from 1 to 32767. If you omit W, the printer width is 80.

The expression:

```
DOSX
```

in the BAT file would start DOSX with 4 variables, 2000-element vectors, and a printer width of 80. The expression:

```
DOSX S2 B4000 W120
```

would start DOSX with 2 shared variables, 4000-element buffers, and a printer width of 120.

### "Abort, Retry, Ignore"

If an operation tries to access a drive that doesn't have a diskette in it, DOS displays a descriptive message, and prompts you to "Abort, Retry, Ignore" the operation.

If you want the operation to proceed then insert the proper diskette and reply R (Retry).

Replies of A (Abort) or I (Ignore) both terminate the DOSX command and cause a result with an rc of 16.

## A simple session using DOSX

```
      710 ⎕svo 'dosx'          Offer variable dosx.

      ⎕svo 'dosx'
2                              Indicates the variable is shared. A 1
                               indicates DOSX wasn't started.

      dosx←'help'              Result of the help command is a list
                               of commands. An empty vector or a
                               vector of blanks has the same result.

      dosx
00 00;
call ;p
chdir fs
close
create fs
dir fs
drives
erase fs
help
mkdir fs
openro fs
openrw fs
peek i,c
poke i;d
read i,c,t
rename fs,nfs
rmdir fs
select a:
settrtab t;d
size
translate t;d
version
write i,t;d
      dosx←'rubbish'
      dosx
07 00;                         7 indicates invalid command. Error
                               codes are at the end of this chapter.

      dosx←'create test.tst'
                               Create a file on the default drive.

      dosx
00 00;                         No error.
```

```
      dosx←'write 0,1;test'
```
Write '*test*', translated to ASCII as indicated by the 1, to location 0.

```
      dosx
00 00;
```

```
      dosx←'write ;data'
```
Write '*data*' at the current location (the end).

```
      dosx
00 00;
```

```
      dosx←'read 0,4,2'
```
Read 4 characters (ASCII to APL).

```
      dosx
00 00;test
```

```
      dosx←'read'
```
Read 4 characters from the current location, with current translation.

```
      dosx
00 00;data
```

```
      dosx←'read 0,8'
```
Read 8 characters from location 0.

```
      dosx
00 00;testdata
```

```
      dosx←'close'
```
Close the file. Note that it is not necessary to read the result.

```
      dosx←'erase test.tst'
```
Erase the file.

```
      dosx
00 00;
```

### DOSX commands

DOSX supports four groupings of commands: DOS file commands, DOS function calls, memory commands (peek and poke), and call of user-written machine-language programs.

Parallel printers are accessed as files with the reserved device names of PRN, LPT1, LPT2, and LPT3.

The *help* command returns a list of all the commands and their syntax. An empty vector or a vector of blanks is treated as a *help* command.

## DOS FILE COMMANDS

DOSX treats a DOS file as a character (byte) vector. Data is read from the file by specifying a 0-origin index, a count, and a translation. Data is written to the file by specifying a 0-origin index, a translation, and the data. The write count comes from the length of the data being written. Data can be appended (catenated) to the end of a file.

DOS files are identified with a file specification (filespec) of the form:

```
[d:][path]filename[.ext]
```

d: An optional letter that identifies the drive (for example, *a* or *b* or *c*). If the drive is not specified, the default drive is used.

path An optional specification of the path to a directory. If path is not specified, the current directory is used.

filename A DOS filename (up to eight characters).

ext An optional DOS file name extension (up to three characters).

Two special characters, *?* and ***, can be used within a filename and extension. *?* indicates that any character can occupy that position. *** indicates that any character (or no character) can occupy that position and all remaining positions in the field.

### Translation

Several commands take a parameter specifying a translation. Five translations are supported.

0 - no translation

1 - APL to ASCII

2 - ASCII to APL

3 - User-defined table (see *settrtab* command)

4 - User-defined table (see *settrtab* command)

5 - Printer (Epson or equivalent) graphic support for APL characters

The APL-to-ASCII and ASCII-to-APL mappings are symmetric. The 95 ASCII graphics map to their corresponding APL graphics. ASCII BS, LF, and CR map to APL BACKSPACE, LINEFEED, and NEWLINE (`⎕av[⎕io+ 158 159 156]`). All other values map to `⎕av[255]`.

The mapping between ASCII CR and APL NEWLINE can cause problems. The ASCII CR often means "return to left margin" and is accompanied by a LF to get to a new line. Sometimes the ASCII LF is used to get to a new line. For a particular application, an adjustment of the translate tables can smooth out these anomalies.

### Current index, count, and translation

A DOS file is treated like a character vector. Reads and writes specify an index (0-origin) and a count to indicate the part of the file affected by the operation and a translation. Reads and writes can elide these parameters. If an argument is elided, then the current value is used. The current value is initialized as follows:

| | Index | Count | Translation |
|---|---|---|---|
| *openro* | 0 | 128 | 0 |
| *openrw* | size of the file (ready for append) | 128 | 0 |
| *create* | 0 | 128 | 0 |

A successful read or write updates the current index by adding the count for the operation to the current index (ready for a sequential operation). A successful read or write updates the current count and current translation if they are explicitly provided. An unsuccessful operation has no effect.

### Open file for read only

*openro* filespec

Opens a file for read only. Writes are not allowed. The argument is a DOS filespec. If the file does not exist, an error code is returned.

```
dosx←'openro test'

dosx←'openro b:test.bat'

dosx←'openro c:\my\private\data.tst'
```

### Open file for read and write

*openrw* filespec

Opens a file for read and write. The argument is a DOS filespec. If the file does not exist, an error code is returned.

```
dosx←'openrw b:test.src'

dosx←'openrw autoexec.bat'
```

**Create a file**

*create* filespec

Creates a file. The file is opened for read and write. If the file already exists, an error code is returned.

```
dosx←'create brand.new'
```

**Read data from the file**

*read* index,count,translation

Reads data from the previously opened file. The parameters specify the 0-origin index to start the read, the count of characters to read, and the translation. An elided parameter indicates that the current index, count, or translation is to be used.

| | |
|---|---|
| `dosx←'read 100,200,2'` | Read 200 characters from index (origin 0) 100 with translation to APL. |
| `dosx←'read ,1000'` | Current index, 1000 count, current translation. |
| `dosx←'read 700'` | 700 index, current count and translation. |
| `dosx←'read'` | Current index, count, and translation. |

**Write data to the file**

*write* index,translation;data

Writes data to the previously opened file. The arguments are the location to write to, the translation, and the data. The data written is the data from the semicolon to the end of the vector. A write can extend the length of a file. A write must start at a location in the file or at the end; it cannot start at a location beyond the end.

| | |
|---|---|
| `dosx←'write 300,1;more'` | '*more*' is written in ASCII at index 300. |
| `dosx←'write ;',data` | *data* is written at the current index with the current translation. |

**Size information**

*size*

Returns the size of the file in bytes, the current index, the current count, and the current translation.

```
      dosx←'size'
      dosx
00 00;1600 320 80 2
```

File is 1600 characters in length. Current index is 320. Current count is 80. Current translation is 2.

**Close the currently opened file**

*close*

Closes (unties) the file. A file must be closed before another file can be opened with this variable. A return code and subcode of 00 02 indicate the file was erased before it was closed.

```
      dosx←'close'
```

**Directory list**

*dir* filespec

Returns a list of all files that match the filespec in the argument. The result is a vector of file names and file information, delimited by carriage returns, selected by the file specification.

Each entry in the result contains the filename (name and extension), the file size, and the timestamp (⎕*ts* format) of when the file was created. A directory is indicated by a \ after the filename. Size information may not be correct for open files.

A directory list may exceed the limit on the size of the shared variable (the B parameter when DOSX was started). The last entry is followed by a period. If the result of a *dir* command doesn't end in a period, then another *dir* command (with no parameter) is required to get more of the result.

The DOSX *dir* command uses the DOS directory function call and is not identical to the DOS DIR command. In particular, the DOSX *dir* argument must be fully specified. For example, the DOS command DIR B: lists all files on drive B, whereas the DOSX *dir* command must have a parameter of *b*:*.* to return all files.

```
      dosx←'dir *.*'
```

All files in the current directory of the default drive.

```
      dosx←'dir b:*.*'
```

All files in the current directory of drive B.

| | |
|---|---|
| *dosx←'dir adir'* | Information about *adir*. |
| *dosx←'dir adir*.*'* | Information about all files in directory *adir*. |
| *dosx←'dir *.bat'* | All files in the current directory on the default drive with an extension of BAT. |
| *dosx←'dir'* | More, if any, files from an earlier request. |

**Erase file**

*erase* filespec

Erases the specified file. The ? and * characters are not allowed in the filespec.

**DOS does not protect you from erasing a file that is open. You can erase a file that is opened with the same DOSX variable, with another DOSX variable, or an APL file tied by the SHARP APL File System. Erasing a file and then continuing to use it has unpredictable results. Avoid erasing files that are opened by any part of your system.**

```
dosx←'erase test.tst'
```

**Rename file**

*rename* filespec,filespec

Renames the file specified in the first parameter to the name in the second parameter. The ? and * characters are not allowed in either parameter.

```
dosx←'rename a.bat,b.bat'
```

**Translate data**

*translate* translation;data

Translates the data according to the translation option and returns the translated data in the result.

| | |
|---|---|
| *dosx←'translate* 1;',⎕*av* | Result is APL ⎕*av* translated to ASCII. |

**Set translate table**

*settrtab* translation;data

Sets the indicated translate table to the values in **data** (which must contain 256 elements). Any translate table can be set, but tables 3 and 4 have been reserved for user tables.

Table 1 is used by DOSX in interpreting commands; changes to Table 1 must not affect the translation of characters that appear up to and including the semicolon in a DOSX argument.

```
dosx←'settrtab 3;',mytable
```

If you set table 3 to *mytable*, and write *mydata* using table 3, the data written is *mytable*[⎕av⍳*mydata*].

You can read a translate table by translating ⎕av. For example, to read translation 3:

```
dosx←'translate 3;',⎕av
⎕←6↓dosx
```

## PRINTER SUPPORT

Parallel printers (for example, the IBM parallel printer, or the Epson printers) can be driven by DOSX. The DOS reserved device names of PRN, LPT1, LPT2, or LPT3 are used to access a printer much as if it were a file. For example:

```
dosx←'openrw prn'
```

opens the first parallel printer. Writes should use an elided index. Reads do not work.

You can also use the utility functions *pkw/openfx*, *pkw/printfx*, and *pkw/closefx*; see appendix F.

Data can be written using any of the translation options. The simplest translation to use is APL to ASCII.

```
c←⎕av[⎕io+156]          APL NEWLINE

dosx←'write ,1;test',c
                        The NEWLINE is needed to force
                        printing on some printers.
```

Translation 0 is useful in sending control characters and escape sequences to cause special actions at the printer. These are different for different printers, so check your printer manual for the ones you can use. Experiment!

```
⎕io←0

t←⎕av[65 66 67 13 10]
                        ASCII "ABC",RETURN,LINEFEED

dosx←'write ,0;',t Prints "ABC"

dosx←'write ,0;',⎕av[15],t,⎕av[18],t
                        Write "ABC" in condensed mode
                        (⎕av[15]) and then write it in
                        normal mode (⎕av[18]).
```

Translation 5 is useful for displaying the special APL characters (for example, in displaying APL functions). This translation uses ASCII graphics where appropriate and uses a fairly common graphic escape sequence for displaying the special APL characters.

```
dosx←'write ,5;abc⍺⍳n',c

dosx←'write ,5;',⎕av,c
```

The W start parameter for DOSX specifies a printer width to be used by translation 5. Translation 5 correctly folds at the width specified in the W start parameter, whether or not the printer folds automatically.

Output that moves the print head using other translation tables can cause subsequent folding with translation 5 to be incorrect. NEWLINE will get things synchronized again.

Translations can be intermixed; that is, you can use translation 0, 1, 2, 3, or 5 as appropriate.

The writes do not have to be done directly to the printer. They can be done to a disk file which can be printed later.

**Printing numeric arrays.** The data you give a DOSX *write* command must be character; but you can use APL's ⍕ function to send a numeric array to the printer. For example:

```
dosx←'write ,5,',,(⍕mat),⎕av[⎕io+156]
```

**Displaying a function definition.** You can use DOSX to display a function definition. For example:

```
dosx←'write ,5,',(1 ⎕fd 'foo'),⎕av[⎕io+156]
```

**Serial printer support.** You can use DOSX to run a printer connected to your PC's serial (asynchronous) communication ports.

First, you have to tell DOS that its pseudofile LPT1 is connected to, say, serial port 1. You do so by including the following command in your startup BAT file:

```
mode lpt1:=com1
```

After DOSX is started, and you've shared a variable with it, you can send data to it as if it were a file:

```
dosx←'openrw lpt1'

dosx←'write ,1;',data
```

## DOS FUNCTION CALLS

DOSX provides access to several DOS function calls as described in the IBM PC **Disk Operating System** manual. Access is not provided at the machine-language level, but rather at a higher level that protects system integrity and is more convenient for the APL programmer.

### Select default drive

*select* d

Uses DOS function call D to select a drive as the default drive. The argument is the drive to select, or empty if this is an inquiry. The result is the previous default drive.

```
      dosx←'select'          Inquire which drive is the default
                             drive.

      dosx
00 00;c:

      dosx←'select b:'       Set default drive to b.

      dosx
00 00;c:                     Was c.
```

### Number of drives

*drives*

Returns the number of drives in the system. It uses DOS function E.

```
      dosx←'drives'

      dosx
00 00;2                      There are two drives.
```

### Version and DOSX information

*version*

Returns the version of the disk operating system and DOSX start parameter information. It uses DOS function call 30.

```
      dosx←'version'

      dosx
00 00;version=2.00,s=4,b=2000,w=80
```

**Make a subdirectory**

*mkdir* filespec

Creates a subdirectory. It uses DOS function call 39 and is similar to the DOS MKDIR command. The argument is the directory to be created.

```
dosx←'mkdir c:\level2'
```
Creates directory *level2* in the root directory.

**Remove a directory**

*rmdir* filespec

Removes a directory. It uses DOS function call 3A and is similar to the DOS RMDIR command. A directory must be empty before it can be removed.

```
dosx←'rmdir c:\level2'
```
Removes directory *level2*.

**Change directory**

*chdir* filespec

Sets the current directory. It uses DOS function call 3B and is similar to the DOS CHDIR command. The command changes the current directory of the specified or default drive. If there is no argument, then no change is made. The result is the previous current directory.

```
dosx←'chdir c:\level1'
```
Sets new current directory for drive C. Result is previous setting.

```
dosx←'chdir \level1
```
Sets new current directory for default drive. Result is previous setting.

```
dosx←'chdir'
```
Current directory of default drive is unchanged and result is current directory of default drive.

## MEMORY COMMANDS (PEEK AND POKE)

You can use DOSX to read and write memory. The *peek* command is used for reading, and the *poke* command for writing. Incorrect use of *poke* can cause serious problems. *peek* and *poke* do not allow default arguments.

### Read from memory

*peek* index,count

The index is the 0-origin index in memory and the count is the number of bytes to read.

```
dosx←'peek 0,100'   Reads the first 100 bytes of memory.
```

### Write to memory

*poke* index;data

The index is the 0-origin index in memory at which to write the data. Do not try this unless you know exactly what you are doing.

## CALL USER-WRITTEN MACHINE-LANGUAGE PROGRAM

*call* ;argument

Executes a machine-language program. The machine-language program can be written directly in machine language ("hand-coded"), or can be easily derived from an EXE file built by the DOS LINK program from the object output of the DOS assembler. The machine-language program can be debugged using DOS DEBUG.

### The *call* argument

The *call* argument to DOSX is a character vector defining initial values for machine registers AX, BX, CX, and DX; a machine-language program; and data arguments for the program.

The argument is loaded into the DOSX buffer and all registers are initialized as follows:

- The AX, BX, CX, and DX registers are loaded from the first four words (a word is two characters). 8086 architecture interchanges the two characters when they are loaded. These values can be immediate arguments to the program or the addresses of data.
- The CS, DS, ES, and SS segment registers are set to address the start of the *call* argument (that is, they address the first byte of the AX register argument).
- The SI and DI registers address the end of the *call* argument.
- The SP and BP registers address the end of the DOSX buffer. The area between SI and SP is the work area that can be used to build a result and for the stack. The size of the DOSX buffer is set by the B start parameter.
- DOSX's CS and IP (necessary for the return) are stacked.
- The Instruction Pointer (IP) is set to address CS+8 (that is, just past the DX register argument).

Figure 11-1 is a memory map of the environment at entry to the *call* program.

The program must return by loading the CS and IP stacked at entry.

**Figure 11-1**

### The *call* result

The *call* result is a character vector containing:

- The usual six-character header (00 00;);
- The values of the 12 machine registers and the flag register, in the order AX, BX, CX, DX, CS, DS, ES, SS, SI, DI, SP, BP, FLAGS; and
- The data between SI and DI when the program returned.

### Writing and testing a machine-language program

You can hand-code the program and pass it directly to *call*. Except for trivial problems this is difficult to do, and nearly impossible to debug.

### Using the assembler

You can use the DOS assembler to write your program. The DOS ASSEMBLER and LINK programs will give you an EXE file of your program. You must use the DOS program EXE2BIN to convert the EXE file to a format suitable for DOSX *call*.

For EXE2BIN to work properly, your assembler source END card must not specify a start address.

You should avoid requiring any segment fixups if at all possible, as this allows the resulting BIN file to be loaded anwhere. If segment fixups are required, you will have to provide the absolute memory address of where the program will be loaded as a parameter to the EXE2BIN program. This is the value of CS at entry to a DOSX *call* program (note that this may change if anything changes in the APL startup).

The BIN file can be read into your workspace using DOSX and can then be passed to the DOSX *call* command.

In general, the program will not contain its arguments. The arguments will be catenated on when the program is given to DOSX *call*.

### Using DEBUG and CALLTEST

The APL and DOSX environment is not very amenable to debugging a machine-language program. A distributed utility program, CALLTEST.EXE, provides the same environment to your program as DOSX, but can do so under control of DEBUG. When your program works correctly with CALLTEST, it will work with DOSX.

CALLTEST takes three start parameters:

P specifies the file containing the program to test. The default is TEST.BIN.

D specifies the file containing the arguments for the program. The data from this file is read and catenated to the program. If the file does not exist or is empty, CALLTEST assumes there is no data. The default is TEST.D.

R specifies the file to which the result is written. This result is the same result that would be returned by DOSX *call*, except that the six-character header is not included. If the file does not exist, it is created. If it exists, the result is written at the end. The default is TEST.R.

When CALLTEST is started, it displays information that will be useful in debugging your program. One such piece of information is "stop address=...". If you run CALLTEST under DEBUG, the first DEBUG stop is after CALLTEST is loaded but before CALLTEST has loaded your program and data; so you can't yet use DEBUG to display your program or set stops. Setting a stop at the stop-address will allow CALLTEST to load your program and data and then stop again, allowing you to display your program or to set stops. You will have to run CALLTEST a first time to learn the stop-address.

### CTOH example

Perhaps the best way to learn how to use DOSX *call* is to study and experiment with distributed files CTOH.ASM and CTOH.BIN.

CTOH.ASM is assembler source for a program that converts its argument to a hexadecimal character representation. This program is commented and is intended to illustrate many useful ideas, so please study it carefully (it is listed at the end of this section).

If you have the assembler; assemble CTOH.ASM to get CTOH.OBJ; link CTOH.OBJ to get CTOH.EXE; and apply EXE2BIN to get CTOH.BIN. File CTOH.BIN is distributed in case you do not have an assembler.

You can use DOSX to create one or more files containing test data. For example, use DOSX to create file TEST1.D and write □*av* to it with no translation.

You can then use CALLTEST as follows:

```
A>calltest pctoh.bin dtest1.d rtest1.r
```

Use DOSX to examine file TEST1.R.

You can use DOSX to read CTOH.BIN into a variable and can then pass it to the DOSX *call* command. For example, if variable *dosx* is shared with DOSX and variable *ctoh* contains the program, then:

```
      dosx←'call;',ctoh,⎕av
      dosx
00 00;000102 ......feff
```

## CTOH.ASM listing

```
; convert character string to apl hex for printing
; this routine is called using the dosx call function
; the data to be converted is catenated to end of code
; the result is
;   the dosx result code 00 00;
;   registers ax,bx,cx,dx,cs,ds,es,ss,si,di,sp,bp,flags,
;   the data between ds:si and ds:di
; for example an argument of quadav produces a result of
; '00 00;',ax,bx,...sp,flags,      32 bytes rc and registers
; '00010203...feff'                512 bytes apl hex data
; if the argument and result don't fit in the buffer
; the result is the result code and registers
ctoh    proc  far
ctseg  segment byte public ;all segment registers point here
        assume cs:ctseg,ds:ctseg,ss:ctseg,es:ctseg
; linkeditor will complain about no stack segment, let it
axval   dw    string   ;ax is address of string
; si and di point to end+1 of string
bxval   dw    0
cxval   dw    0
dxval   dw    0
entpt:                 ;entry point is always address 8
        push  si       ;save source index
        mov   si,ax    ;start of data string
        mov   cx,di    ;end of string
        sub   cx,ax    ;length of string
        jz    done     ;no string
        mov   ax,cx
        add   ax,ax    ;length of result (twice arg length)
        add   ax,di    ;end of result
        add   ax,200   ;room for stack
        cmp   ax,sp
        jae   done     ;no room, return empty result
        xor   ax,ax
next:
        lodsb          ;get source byte into al
        push  ax       ;save byte
        mov   bx,ax
        shr   bx,1     ;shift top half byte down
        shr   bx,1
        shr   bx,1
        shr   bx,1
        mov   al,table[bx] ; get apl char for first half
        stosb          ;store value in result
        pop   bx       ;get data byte
```

```
        and   bx,0fh    ;remove top half byte
        mov   al,table[bx] ; get apl char for second half
        stosb           ;store value in result
        loop  next      ;decrement cx and jump if not zero
done:                   ;conversion complete
;di now points to end of result
        pop   si
;si points at start of result
        ret             ;return to dosx
table   db    8ch,8dh,8eh,8fh,90h,91h,92h,93h ;apl 0-7
        db    94h,95h,56h,57h,58h,59h,5ah,5bh ;apl 8,9,a-f
string:                 ;data starts here
ctseg   ends            ;end segment
ctoh    endp            ;end procedure
        end             ;start address 0  for exe2bin
```

## SUMMARY OF COMMANDS

```
call ;p
chdir fs
close
create fs
dir fs
drives
erase fs
help
mkdir fs
openro fs
openrw fs
peek i,c
poke i;d
read i,c,t
rename fs,nfs
rmdir fs
select a:
settrtab t;d
size
translate t;d
version
write i,t;d
```

fs - filespec
nfs - new filespec
i - index (0-origin)
c - count
d - data
t - translation option
a - drive letter

## RETURN CODES

| Return code | Meaning |
|---|---|
| 00 | Command executed successfully |
| 01 | File not found |
| 02 | File already exists |
| 03 | A file is already open |
| 04 | Index error (read or write outside of the file) |
| 05 | Invalid count |
| 06 | Invalid argument |
| 07 | Invalid command |
| 08 | Length error |
| 09 | Command not implemented |
| 10 | No file open |
| 11 | System error |
| 12 | No space, disk full |
| 13 | Write to read-only file |
| 14 | DOS access error |
| 15 | Rename failed |
| 16 | No disk in the drive. Response of Ignore or Abort to DOS Abort,Retry,Ignore message. |
| 17 | Invalid filespec |

### Subcodes

The subcode is in hexadecimal (that is, has values from 00 through FF). Subcodes are DOS error returns and are documented in the manuals IBM provides with your version of DOS.

# Chapter 12: ASYNCHRONOUS COMMUNICATIONS AUXILIARY PROCESSOR (ACOM)

This chapter describes the Asynchronous Communications Auxiliary Processor (ACOM). An APL task can use ACOM to communicate with other systems or devices through your personal computer's primary and secondary asynchronous ports. For example, ACOM can be used to communicate with a remote timesharing system, another PC, a printer, or a plotter.

This chapter is quite technical, and assumes some familiarity with RS232 asynchronous communication protocol and terminology. With or without that background, you are encouraged to learn how to use ACOM through experimentation; you can communicate with many different systems and devices using ACOM. Chapter 5 **Communicating with a SHARP APL/370 system** discusses in detail how to use ACOM for communicating with a SHARP APL timesharing service.

Workspace 10 *acom* on most SHARP APL/370 systems contains several functions useful in working with ACOM.

## Shared variables

ACOM can share up to six variables. Each variable is independent of the others. This allows several ACOM commands to be used simultaneously. For example, the command to receive data on the primary port could be requested on one variable, while another variable was waiting for a transmission request to be completed on the same port.

ACOM's processor number is 700 (unless you specify another value with the U start parameter).

To share a variable with ACOM, you must first offer to share a variable with it; for example:

```
      700 ⎕svo 'acom'      Offer to share a variable named acom
```

And then check that the offer has been accepted (that is, that ACOM is running):

```
      ⎕svo 'acom'
2                          2 indicates variable is shared
```

To request an ACOM command, set the shared variable to a **command string** (cs). When you use the variable, you will find a **result string** (rs).

Both command and result are character strings: 16-character **headers** followed by any data.

ACOM sets access control so that each variable is fully interlocked.

## ACOM commands

**Start command**
initializes a port for transmission or reception.

**Transmit/Receive command**
transmits and/or receives data.

**Transmit Break command**
terminates any transmission, and then sends a BREAK signal.

**Stop command**
stops all activity on the port.

**Inquire command**
returns status information about a port.

**Set/Read Frequency Table command**
sets or returns the character frequency table used in translation 4.

An index origin (`⎕io`) of 0 is assumed throughout this chapter.

## ACOM start parameters

ACOM must be started after SVP and before APL.

**Ln**
specifies that asynchronous port **n** is attached to an I.P. Sharp linedriver. **n** may be 0 or 1 or blank (meaning both ports). If you omit L, you indicate that neither port is attached to a linedriver.

**Qn**
indicates whether ACOM is to display its start message. **n** may be 0 (specifying that the message is not to be displayed) or 1. If you omit Q, the message is displayed.

**Un**
specifies ACOM's processor number. **n** may be any value from 2 to 999. If you omit U, ACOM's processor number is 700.

## START COMMAND

The **Start** command indicates to the asynchronous port that the APL task is ready to transmit and/or receive data. This command also specifies several parameters for the communication: autoanswer timeout, communication rate, parity, the number of stop and data bits per character, and flow control.

### Command string

The command string comprises only a 16-character header:

**cs**[0]
is ⎕*av*[0], specifying the Start command.

**cs**[1]
specifies the port:

| value | meaning |
|---|---|
| ⎕*av*[0] | primary port |
| ⎕*av*[1] | secondary port |

**cs**[2 3]
specifies the autoanswer timeout. If ACOM can't establish a path or "handshake" with the other end, you want to prevent your program from hanging. So you set a timeout period. The expression for the timeout, in seconds, is: (256⊥⎕*av*⍳**cs**[2 3])÷10. This timeout is not used if **cs**[9] is set on.

If DSR is not raised before the autoanswer timeout expires, a line drop error is indicated in **rs**[2 3]. If no autoanswer timeout is specified, ACOM will wait indefinitely for DSR to be raised (unless **cs**[9] is ⎕*av*[1]).

A value of ⎕*av*[0 0] specifies that no timeout is in effect. The maximum allowed value is ⎕*av*[127 255]. A value of ⎕*av*[0 100] specifies a ten-second timeout.

**cs**[4]
specifies the communication rate:

| value | meaning |
|---|---|
| ⎕*av*[0] | 300 bits per second |
| ⎕*av*[1] | 1200 bits per second |
| ⎕*av*[2] | 2400 bits per second |
| ⎕*av*[3] | 4800 bits per second |
| ⎕*av*[4] | 9600 bits per second |

cs[5]
specifies parity:

| value | meaning |
|---|---|
| ⎕av[0] | none |
| ⎕av[1] | odd |
| ⎕av[2] | even |
| ⎕av[3] | mark |
| ⎕av[4] | space |

cs[6]
specifies the number of stop bits per character:

| value | meaning |
|---|---|
| ⎕av[1] | 1 stop bit |
| ⎕av[2] | 2 stop bits |

cs[7]
specifies the number of data bits per character:

| value | meaning |
|---|---|
| ⎕av[5] | 5 characters |
| ⎕av[6] | 6 characters |
| ⎕av[7] | 7 characters |
| ⎕av[8] | 8 characters |

cs[8]
specifies flow control. When you are receiving data, and the buffer is becoming full, RS232 has a standard procedure for suspending the transmission of data. XON/XOFF flow control is normally used. CTS/RTS (Clear to Send/Request to Send) flow control is usual only for Teletype devices.

| value | meaning |
|---|---|
| ⎕av[0] | no flow control |
| ⎕av[1] | XON/XOFF flow control |
| ⎕av[2] | CTS/RTS flow control |
| ⎕av[3] | both XON/XOFF and CTS/RTS flow control |

cs[9]
indicates whether DSR is to be monitored or ignored:

| value | meaning |
|---|---|
| ⎕av[0] | DSR is to be monitored |
| ⎕av[1] | DSR is to be ignored: line drops are not reported in any return code; lack of DSR does not affect any command |

cs[10]
indicates whether a Break received during transmission is to be processed or ignored:

| value | meaning |
|---|---|
| □*av*[0] | Break is to be processed: if a transmission is in progress, it is aborted and its return code indicates that a Break was received; if transmission is not in progress, the next Rx return code indicates that Break was received |
| □*av*[1] | Break is to be ignored; that a Break was received is indicated in the next return code |

**cs**[11] through **cs**[15]
must be □*av*[0].

### Result string

The result string comprises only the 16-character header:

**rs**[0]
is □*av*[0], indicating the Start command.

**rs**[1]
indicates the port.

**rs**[2 3]
is the first return code; see **Return codes**.

**rs**[4 5]
is the second return code; see **Return codes**.

**rs**[6]
returns the modem signals; see **Modem signals**.

**rs**[7] through **rs**[15]
are □*av*[0].

### Usage

The command is completed and the result string returned when either the autoanswer timeout has expired, or Data Set Ready (DSR) is raised (that is, the other end indicates that it is ready for communication).

## TRANSMIT/RECEIVE COMMAND

The **Transmit/Receive** command transmits or receives data. Data to be transmitted follows the 16-character **cs** header; received data follows the 16-character **rs** header.

### Command string

The command string comprises the 16-character header and data for transmission:

**cs**[0]
is □*av*[1], specifying the Transmit/Receive command.

**cs**[1]
specifies the port:

| value | meaning |
|---|---|
| □*av*[0] | primary port |
| □*av*[1] | secondary port |

**cs**[2]
specifies transmit and receive translation:

□*av*[0]
indicates no translation.

□*av*[1]
indicates APL-ASCII translation; received characters are not resolved.

□*av*[2]
indicates six-bit transparent translation.

□*av*[3]
indicates APL-ASCII translation; received triplets are resolved.

□*av*[4]
indicates character-frequency-sensitive transparent translation.

For more information about translation, see below.

**cs**[3]
specifies the integrity checksum (ICS) option in effect:

| value | meaning |
|---|---|
| □*av*[0] | no checksum |
| □*av*[1] | ICS on transmit: ICS inserted before last character |
| □*av*[2] | ICS on receive: ICS precedes last character |
| □*av*[3] | ICS on both transmit and receive |

Workspace 10 *acom* on SHARP APL/370 systems contains functions for inserting and checking the ICS. You do not need these functions on your PC, as ACOM does it for you, but they are needed by the program at the other end of the line.

**cs**[4]
specifies the receive special character, that is, the character that marks the end of the received string (usually BELL, indicated by ⎕*av*[7]). ⎕*av*[0] indicates that there is no special character; any other value is the transmission code of the special character (this is the value before the translation indicated by **cs**[2]).

**cs**[5 6]
indicates the receive intercharacter timeout. This timeout prevents your program from hanging if no characters are being received (presumably due to a line fault). ⎕*av*[0 0] indicates that there is no timeout.

The expression for the number of seconds for the timeout is: (256⊥⎕*av*⍳**cs**[5 6])÷10. The maximum is ⎕*av*[127 255].

**cs**[7 8]
specifies the receive first-character timeout. If you are expecting to receive data, you can set a timeout for the first character; this may be distinguished from **cs**[5 6], which measures the interval between successive characters. ⎕*av*[0 0] indicates that no timeout interval is set. The expression for the number of seconds for the timeout is: (256⊥⎕*av*⍳**cs**[7 8])÷10. The maximum is ⎕*av*[127 255].

**cs**[9 10]
specifies the receive count, that is, the maximum number of transmission codes allowed in the receive buffer. ⎕*av*[0 0] indicates that this command is for transmit only. The expression for the count is: 256⊥⎕*av*⍳**cs**[9 10]. The maximum is ⎕*av*[127 255].

**cs**[11]
indicates whether the receive buffer is cleared:

⎕*av*[0]
indicates that the buffer is not to be cleared.

⎕*av*[1]
indicates that the buffer is to be cleared when a Receive command starts.

**cs**[12] through **cs**[15]
must be ⎕*av*[0].

**cs**[16] onward
are data characters for transmission. The maximum number of characters that can be transmitted is 1000 **after translation**. If there are no data characters, this is a receive-only command.

### Result string

When transmission and reception are complete, or when an error has occurred, the result string comprises the 16-character header and up to 1000 characters of received data:

**rs**[0]
is □*av*[1], indicating the Transmit/Receive command.

**rs**[1]
indicates the port.

**rs**[2 3]
is the first return code; see **Return codes**.

**rs**[4 5]
is the second return code; see **Return codes**.

**rs**[6]
returns the modem signals; see **Modem signals**.

**rs**[7] through **rs**[15]
are □*av*[0].

**rs**[16] onward
are the received data characters.

### Translation

The translation ACOM is to apply to transmitted or received data is specified in **cs**[2].

**No translation** is indicated by *av*[0]. No translation is useful if your software needs to handle the transmission codes directly.

**APL-ASCII translation** is indicated by □*av*[1] or □*av*[3]. Whichever type is specified, ASCII data is transmitted or received on the line and APL data is referenced and set in the shared variable. The translation used is the typewriter-pairing APL-ASCII overlay described in the **SHARP APL Reference Manual** and **SHARP APL Pocket Reference**.

If you specify □*av*[1], compound APL characters are transmitted as triplets (character, BACKSPACE, character), but received triplets are not resolved into a single element of □*av*. (The Session Manager resolves such triplets on display.) Received pairs of ASCII CR LF are resolved into APL NEWLINE.

If you specify □*av*[3], compound APL characters are transmitted as triplets (character, BACKSPACE, character), and received triplets are resolved into a single element of □*av* using the Session Manager's overstrike table. (For more information about the overstrike table, see appendix B **Customizing the keyboard and character set**.) Received pairs of ASCII CR LF are resolved into APL NEWLINE.

**Transparent translation** is indicated by `⎕av[2]` or `⎕av[4]`.

If you specify `⎕av[2]`, data is transmitted and received in six-bit transmission codes. Although data is manipulated in eight-bit characters in the workspace, a communication line can usually only be used to transmit six bits of data in each character. Six-bit transparent translation facilitates moving blocks of eight-bit characters between your PC and another computer (for example, a timesharing system with which you are exchanging data). You set eight-bit characters in the ACOM variable, and ACOM transmits all the information, but in six-bit units. A program at the other end must combine the six-bit codes back into the original eight-bit character data. Similarly, the other end can send you the six-bit codes that ACOM reassembles and passes to your program as eight-bit character data.

Workspace 10 *acom* on SHARP APL/370 systems contains functions for converting between eight-bit characters and the equivalent data in six-bit transmission codes. You do not need these functions on your PC, as ACOM does it for you, but they are needed by the program at the other end of the line.

If you specify `⎕av[4]`, characters are transmitted and received in either one or two transmission codes, depending on their placement in a character frequency table. The first 93 elements of the frequency table are sent and received as a single code each. The remaining characters are sent and received as two codes each. This translation is particularly efficient for most text- and data-entry applications. The frequency table is controlled by the Set/Read Frequency Table command.

## Usage

To transmit only, set the receive count (`cs[9 10]`) to zero.

To receive only, set `cs` without any data after the 16-character header.

Whenever there is not an outstanding Receive from a Transmit/Receive command, the receive buffer is limited to 128 characters.

If ACOM's buffer is smaller than the receive count (`cs[9 10]`), the buffer size limits the number of characters received.

If the receive checksum is selected but the receive count expires before the special character, `cs[4]`, is received, a checksum error is reported in the return code.

The checksum algorithm is: `32+32 32 32⊤+/⎕av⍳16↓¯1↓cs`.

ACOM has one buffer for each port. This buffer is dynamically shared between data waiting for transmission and data received, but not yet set in a shared variable. The number of characters that can be transmitted is 1000 minus the outstanding receive count

(**cs**[9 10]). For example, if a receive with count **cs**[9 10] of 500 has been started, the maximum number of transmited characters possible is 500 (1000-500). The space available in the AP buffer for a receive is 1000 minus the count of data in outstanding transmission. For example, if a transmission of 400 characters is in progress, the maximum number of characters in **rs** is 600 (1000-400).

## TRANSMIT BREAK COMMAND

The **Transmit Break** command clears the transmit buffer and sends a BREAK signal. Any transmission in progress is aborted.

When your program hangs, or when you want to stop receiving output, you can use Transmit Break to transmit a break to clear things out. It is equivalent to pressing the BREAK or ATTN key on a terminal.

### Command string

The command string comprises only the 16-character header:

**cs**[0]
is □*av*[2], specifying the Transmit Break command.

**cs**[1]
specifies the port:

| value | meaning |
|---|---|
| □*av*[0] | primary port |
| □*av*[1] | secondary port |

**cs**[2]
indicates whether the receive buffer is cleared:

□*av*[0]
indicates that the buffer is not to be cleared.

□*av*[1]
indicates that the buffer is to be cleared when the Break is transmitted.

**cs**[3] through **cs**[15]
must be □*av*[0].

### Result string

**rs**[0]
is □*av*[2], indicating the Transmit Break command.

**rs**[1]
indicates the port.

**rs**[2 3]
is the first return code; see **Return codes**.

**rs**[4 5]
is the second return code; see **Return codes**.

**rs**[6]
returns the modem signals; see **Modem signals**.

**rs[7] through rs[15]**
**are ⎕*av*[0].**

## STOP COMMAND

The **Stop** command drops the line; specifically, it sets Data Terminal Ready (DTR) and Ready to Send (RTS) low, and stops the receiver. Any outstanding commands on the port are terminated.

### Command string

**cs**[0]
is □*av*[3], specifying the Stop command.

**cs**[1]
specifies the port:

| value | meaning |
|---|---|
| □*av*[0] | primary port |
| □*av*[1] | secondary port |

**cs**[2] through **cs**[15]
must be □*av*[0].

### Result string

**rs**[0]
is □*av*[3], indicating the Stop command.

**rs**[1]
indicates the port.

**rs**[2 3]
is the first return code; see **Return codes**.

**rs**[4 5]
is the second return code; see **Return codes**.

**rs**[6]
returns the modem signals; see **Modem signals**.

**rs**[7] through **rs**[15]
are □*av*[0].

## INQUIRE COMMAND

The **Inquire** command returns information about the current state of the port.

### Command string

**cs**[0]
is ⎕*av*[4], specifying the Inquire command.

**cs**[1]
specifies the port:

| value | meaning |
|---|---|
| ⎕*av*[0] | primary port |
| ⎕*av*[1] | secondary port |

**cs**[2] through **cs**[15]
must be ⎕*av*[0].

### Result string

**rs**[0]
is ⎕*av*[4], indicating the Inquire command.

**rs**[1]
indicates the port.

**rs**[2 3]
is the first return code; see **Return codes**.

**rs**[4 5]
is the second return code; see **Return codes**.

**rs**[6]
returns the modem signals; see **Modem signals**.

**rs**[7] through **rs**[15]
are ⎕*av*[0].

**rs**[16]
returns the modem signals like **rs**[6].

**rs**[17]
indicates whether a transmission is in progress:

| value | meaning |
|---|---|
| ⎕*av*[0] | no transmission is in progress |
| ⎕*av*[1] | transmission in progress |

**rs**[18]
indicates whether a reception is in progress:

| value | meaning |
|---|---|
| □*av*[0] | no reception in progress |
| □*av*[1] | reception in progress |

**rs**[19 20]
indicates the buffer size. The expression for the buffer size is: `256 256 ⊥□av⍳rs[19 20]`.

## SET/READ FREQUENCY TABLE COMMAND

The **Set/Read Frequency Table** command controls the character frequency table used in character-frequency-sensitive transparent translation. (For more information about translation, see **Transmit/Receive command**.

### Command string

**cs**[0]
is ⎕*av*[5], specifying the Set/Read Frequency Table command.

**cs**[1] through **cs**[15]
must be ⎕*av*[0].

**cs**[16] through **cs**[271]
is the character frequency table to be set. If you omit these (and send only the 16-character header), the table is not changed, only returned.

**cs**[16] through **cs**[108]
are the characters to be transmitted or received as single transmission codes.

**cs**[109] through **cs**[271]
are the characters to be transmitted or received as double transmission codes.

### Return string

**rs**[0]
is ⎕*av*[5], indicating the Set/Read Frequency Table command.

**rs**[1] through **rs**[15]
are ⎕*av*[0].

**rs**[16] through **rs**[271]
are the character frequency table, in the format described above.

## RETURN CODES

### First return code

The two characters in the first return code contain 16 bits (boolean values) of information. Each bit indicates whether a particular problem or state exists.

To display the bit values, use this expression:

```
(16⍴2)⊤256⊥⎕av⍳rs[2 3]
```

This displays a vector of 16 1's or 0's. The leftmost value is bit 0, and the rightmost is bit 15.

Only **one** of bits 0 through 11 may be set on. Any combination of bits 12, 13, 14, and 15 may be set on in the return code of a Transmit/Receive command for reception if bit 8, 9, 10, or 11 is also set on.

This is what each bit means when it is set on, that is, to 1:

**Bit 0**
indicates that a Break was received during reception.

**Bit 1**
indicates an error in the command string.

**Bit 2**
indicates a line drop; always 0 if **cs**[9]=⎕*av*[1] in the Start command.

**Bit 3**
indicates abort. The command is aborted because:
- there is already an outstanding command for this port;
- the Start or Stop command was issued during this command operation; or
- a Transmit Break command was issued and this is a Transmit command.

**Bit 4**
indicates a break was received during a transmission; only set if **cs**[14]=⎕*av*[4] in the Start command.

**Bit 5**
is not used.

**Bit 6**
indicates a transmission buffer overflow error; that is, there is not enough room in the buffer for the transmission data.

**Bit 7**
indicates a transmission count expired; that is, the supplied characters have been transmitted.

**Bit 8**
indicates a receive checksum error.

**Bit 9**
indicates a receive timeout expired; that is, one of the timeouts specified in the Transmit/Receive command (**cs**[5 6] or [7 8]) has expired.

**Bit 10**
indicates receive count expired; that is, the number of characters specified in the Transmit/Receive command (**cs**[9 10]) has been reached.

**Bit 11**
indicates a receive special character received.

**Bit 12**
indicates a receive framing error.

**Bit 13**
indicates a receive parity error.

**Bit 14**
indicates a receive overrun error.

**Bit 15**
indicates a receive buffer overflow error: the specified flow control failed to stop the remote transmitter, and the receive buffer overflowed, losing data.

**Second return code**

**rs**[4] indicates the reason for the operation's ending, according to the following values:

□*av*[1]
indicates that the command was completed successfully. If the command was Transmit or Receive, count has expired.

□*av*[2]
indicates that the command failed to complete. **rs**[5] gives the reason, according to the following values:

□*av*[0]
indicates an undefined error.

□*av*[1]
indicates an error in the command string.

□*av*[2]
indicates a line drop. (This value is not possible if **cs**[9]=□*av*[1] in the Start command.)

□*av*[3]
indicates the command was aborted because: there is already an outstanding command on this port that prevents this command from being processed; a Stop or Start command was issued during this command's operation; or the Transmit Break command was issued during this command, and this is a Transmit command.

□*av*[6]
indicates a Transmit buffer overflow: there is not enough room in the buffer for the transmission data.

□*av*[3]
indicates that the Receive command has ended because the Receive special character was received.

□*av*[4]
indicates that the Receive command ended because the specified Receive timeout has expired.

□*av*[5]
indicates that the Receive command ended because there was a checksum error.

□*av*[6]
indicates that the Transmit command was aborted because a Break was received. This code is not possible unless **cs**[10]=□*av*[8] in the Start command.

If the command was Receive and if **rs**[4] was □*av*[1], □*av*[3], □*av*[4], or □*av*[5], **rs**[5] contains receive error information; its value is □*av*[**x**], where **x** is the **sum** of the following:

1
indicates that the Receive buffer overflowed: the specified flow control failed to stop the remote transmitter, and data was lost.

2
indicates a Receive overrun error.

4
indicates a Receive parity error.

8
indicates a Receive framing error.

16
indicates that Break was received.

## MODEM SIGNALS

Modem signals are returned in `rs[6]`, encoded as `a←(8ρ2)⊤rs[6]`:

`a[2]`
indicates that DSR is set.

`a[3]`
indicates that CTS is set.

`a[6]`
indicates that RTS is set.

`a[7]`
indicates that DTR is set.

# Chapter 13: FULL SCREEN MANAGER (FSCR)

It's possible to set up a program so that it generates a display that occupies the entire screen, and permits you to write instructions or replies in any appropriate place on the screen. For example, you might want a program to maintain a journal of transactions. It could display a form, with headings to indicate the various categories of information, and blanks to receive the various entries you supply. Or it might display the information it now has, and let you edit the display to correct it.

It's also useful to be able to assign meanings to the **function keys** at the left of the main keyboard, and use them to select from a menu of options merely by pressing a button. Or you may want to move the cursor to some place on the screen so as to point to a particular region and thereby indicate a choice.

You can do such things with the Full Screen Manager (FSCR), a distributed auxiliary processor whose processor ID is 124.

## Compatibility with AP124

FSCR is functionally equivalent to AP124 distributed with SHARP APL/370. AP124 is used with SHARP APL/370 to manage the screens of IBM 3270 display stations. FSCR manages the PC screen in the same way. In general, a program that runs with AP124 in SHARP APL/370 will also run with FSCR on the PC (unless it exploits the light pen, equipment that is optionally available for the 3270 but not for the PC). The converse is not quite true, because the PC is a bit more tolerant than the 3270; for example, FSCR permits one field to be right next to another, whereas the 3270 requires a horizontal separation of one or two positions between them. Such points of difference between the two environments are listed in the notes at the end of this chapter.

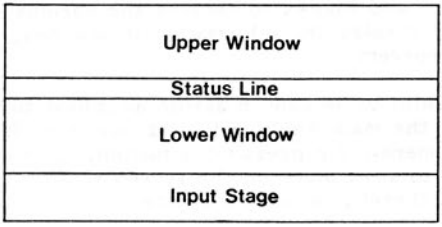
The I.P. Sharp APs (AP124 for SHARP APL/370, FSCR for SHARP APL/PC) are both upwardly compatible with AP124 provided by IBM. That is, a program written for IBM's AP124 will run under the I.P. Sharp version of AP124. The converse isn't necessarily true because the I.P. Sharp version has many enhancements not provided by IBM.

## Contrasting FSCR with the Session Manager

To introduce the concepts that underly FSCR, we first review how things work when you **don't** use it, but rely on SHARP APL/PC's Session Manager. The Session Manager divides your screen into a staging area and a log. You have two windows through which you can see the log. The stage is the lowest portion of your screen. There you prepare an entry that you want APL to execute. You **release** an entry for action by pressing <return>. Each time you release an entry, the Session Manager updates the log by appending your entry and APL's response, and moves the log up so that they're the last items showing in the lower window. Although work on a screen permits some tricks that you couldn't do with typewriter and paper, there's still an underlying similarity to work

at a typewriter on a very long scroll of paper; you might imagine the Session Manager's writing on that scroll as it rolls past your two windows and your movable highlighter, the "peg".

**Figure 13-1: Screen areas**




When you ask FSCR to produce a display, it seizes control of the entire screen. No more status line, no more stage, no more log; for the time being, they all disappear. FSCR replaces whatever the Session Manager had displayed with a display of its own. FSCR works with a succession of pictures--or **screens**--each of which occupies the entire 25×80 area. FSCR is able to display characters anywhere on the screen. It's also able to make parts of the screen brighter, or blinking, or underlined, or reversed, or appear in colour (on a PC equipped for colour).

### Alternation between entry and display: full-screen dialogue

Just as with the Session Manager, FSCR conducts a dialogue: an alternation between displays that FSCR generates and responses that you make from the keyboard. Just as with the Session Manager, FSCR waits while you prepare input until you **release** it. But in full-screen work, while it's your turn, the entire screen is your stage. Instead of writing just in the staging area at the bottom of the screen, FSCR lets you move the cursor anywhere. You write directly on the display itself--in zones provided to receive your input. (FSCR restricts you to writing in parts of the screen that have been declared to be "keyable fields", but when you lay out the screen, you're free to put those wherever seems appropriate.) Before you release the screen, you may write a lot--a whole screenful, even--or you may just press a single key to indicate your response.

While the Session Manager provides only one way to release an entry for processing (pressing <return>), FSCR provides several. You release the screen to FSCR either by pressing <return> or by pressing any of the **function keys**, labelled <F1> to <F10>, at the left of the main keyboard. An application may assign a meaning to each key, so that you signal a particular action by pressing the appropriate key. You may write on the screen before you press a function key, or you may press one immediately, without writing anything.

When you release a screen to FSCR, it has a record of what you did, and can report that. It can report which key you used to

release the screen, where on the screen the cursor was located when you did so, which parts of the screen you altered, and what was written there when you'd completed your changes.

Once a program has caused FSCR to write to the screen, FSCR's display remains visible until APL makes a request for input or output via the Session Manager. That is, until the expression or program which produced the display reaches a normal end, or encounters an untrapped error, or comes to a request for ⎕ or ⍞ input or output. When any of those happens, the normal screen manager takes over, and log and stage reappear. However, FSCR retains the description of the screen, and can still respond to inquiries about it, or, when commanded to do so, can again seize control of the screen and regenerate the former display.

### A simple example

If you haven't yet used FSCR at all, the discussion may make more sense after you take a few moments to run a sample program. Experiment with FSCR by loading distributed workspace *fscrex*. Its latent expression (⎕*lx*) contains an expression so that, when the workspace is loaded, it automatically offers to share two variables with FSCR, and reports whether it has been successful. You should see the message *ctlS & datS shared with FSCR*. (However, if FSCR isn't running or isn't responding, the message contains the word *not*.)

To run the sample program, execute *sample*. (If for some reason you haven't got the workspace *fscrex*, you can key in the program yourself in a few minutes from the description that appears on the next page.)

The sample program displays four fields. They're all in reverse video, to make obvious where on the screen they're located. There's a caption which appears in a single band across the top. The program is written so that the caption is displayed first, and then the other three fields are added all at once.

Of the three remaining fields, two appear side by side below the caption. On the left you'll see a table of names, and on the right a column of numbers. Below these, the fourth field occupies most of the lower part of the screen. It is partly filled by repetitions of the sentence '*This is a free-form text field.* '

When the four fields are visible on the screen, the program executes instructions that put the cursor at the top left corner of the second field, sound the alarm (which emits a beep), and then cause the computer to wait for input from the keyboard. You can move the cursor around, and insert or delete characters at various places. FSCR continues to collect input until you press <return> or any of the function keys at the left edge of the keyboard. At that point, the program asks FSCR for a report of your entries. Since the program doesn't ask for any further display, the screen vanishes, and you're returned to the Session Manager, where you'll see a report that is the program's result.

While the screen is visible, experiment with it. Use the keys at the right labeled ← → ↑ ↓ to move the cursor. Use the <tab> key in both base and shift cases. Use the <Ins> key to switch into and out of insertion mode (and notice the insert indicator ∧ at the bottom right of the screen).

Try overwriting data in each of the fields. Try inserting data in each of the fields, with and without first deleting some characters to make room. Try to insert many characters into field 4, and then delete them.

Finally, to release the screen, press <return> or one of the function keys at the left.

Some points to notice:

- In some fields you can't insert or delete anything.
- In some fields you can write numbers but not letters.
- In some fields you can insert new characters at the beginning of a line without first deleting, while in others you have to delete first to make room.
- In some fields an insertion or deletion is confined to the row you're working on, while in others, work on one row affects the rows that follow.

The report that the sample program returns is a three-element vector of enclosed arrays. The workspace has □*ps* set so that each element is displayed with a frame around it, making it easy to see. The first element is a numeric vector. It shows which key you pressed to release the screen, where on the screen the cursor was at the time, and which of the fields you had altered. The second and third elements show the contents of fields 2 and 3 as they were at the time you released the screen.

### Other experiments with the sample program

The program *sample* merely generates a table of names and a list of numbers, and invokes *smpl* to do the actual work:

```
    ∇ r←sample; numbers; names
[1]   numbers←numbers,+/numbers←0.01×?3ρ100000
[2]   names←'\' separate 'John Jones\Maria Casas\Ken
                                              Nakanishi'
[3]   r←numbers smpl names
    ∇
```

As you can see, *smpl* takes an array of numbers as its left argument and an array of characters as its right argument. Try executing *smpl* using your own data as its arguments. (A complete listing of *smpl* appears at the end of the chapter. Many of its statements consist of FSCR commands; after you've read their

definitions later in this chapter, the listing of *smpl* should make sense.)

### Outline of how you pass instructions to FSCR

FSCR is an auxiliary processor. Your APL workspace communicates by sharing a variable (or a pair of variables) with it. That means that both the Shared Variable Processor (SVP) and FSCR must be started at the same time that you start APL. (For information about starting SVP, FSCR, and APL, see chapter 2 **Getting started**.)

What follows describes the individual steps that transpire as you communicate with FSCR. Probably you'll incorporate these steps into subfunctions, or use the utility functions supplied in distributed workspace *fscr*. That will automate and conceal the mechanism, and spare you from having to write it all out in detail. The following paragraphs give you a general explanation of what's going on under the covers.

- You (that is, your program on your behalf) offer to share a variable with FSCR.
- Your program **sets** the shared variable (that is, assigns a value to it). FSCR uses that value. (See chapter 9 **Shared variables and auxiliary processors** for a discussion of communication by shared variables.) The values that you set tell FSCR how you want the screen to appear, what modifications from the keyboard to permit, or what information you want back.
- FSCR sets the value of the shared variable to report that it has carried out your command (or, if it was unable to carry it out, what sort of trouble prevented it). It also sets one of the shared variables to report where on the screen you wrote, what you wrote there, and what key you pressed to release the screen.
- Your program examines the reports it receives from FSCR and decides what to do next. It may decide to request a new display, or a modification of the one just displayed, or even to continue the display unchanged; some applications display screen after screen, not returning to immediate execution (and hence not returning to the Session Manager) until much later in the work session. Or your program may decide to do something else entirely, or to terminate.

Throughout its use of FSCR, your program alternates between **setting** the shared variable with commands, and then **using** the shared variable to find out what happened. (You may want to look back to chapter 9 to review those concepts.)

### FSCR's internal record of the screen and the screen you see

FSCR maintains (in its own area of storage) tables that describe the appearance of the screen. Programmers usually refer to these as the screen **buffers**. They determine what will be displayed when a display is requested.

Most of FSCR's commands affect those stored tables (rather than the display directly). A program that uses FSCR has to start by setting up those tables. Certain commands cause FSCR to format the current contents of its tables to the screen. If you use the keyboard to alter the screen by inserting or deleting characters, FSCR immediately posts those changes, character by character as they occur, into its internal tables. When you ask to "read what is on the screen", you are really getting a report in what's now in the tables that FSCR maintains. (Indeed, it is possible--if eccentric--to set up the tables and then read what is in them without ever actually displaying their contents to the screen.)

Some commands simply change the data in the buffer tables. Others cause FSCR to generate a new screen display. A display always copies to the screen the data from all of the buffers. Some of that data may have been placed in the buffers much earlier, and remained there unchanged; some of it may have been placed there only a moment before, as part of the same command that elicits the display.

Even when FSCR relinquishes control of the screen, its buffers containing the screen data remain intact. You can still read what is in them, or use them to regenerate the screen. Only when the shared-variable link is broken does FSCR clear its screen buffers.

### Reading, writing, and fields

FSCR divides the screen into **fields**. A field is a rectangular area of the screen (defined by the coordinates of its top left corner) and its shape (that is, the number of rows and columns it occupies). In order to have your program display something on the screen, you must tell FSCR three things:

- Where each field is located (and certain other attributes). FSCR keeps its own record of the specifications you provide.
- What data you want displayed in each field. FSCR also keeps its own record of the data for each field.
- That you want the screen displayed. FSCR then displays the entire screen, including the data for all the fields. (If some fields haven't changed since your preceding display, you don't have to specify them again; at each display, all fields appear, showing the new data where you've supplied it, or repeating the old where you haven't.)

When you use the keyboard to alter or write on a screen display, FSCR permits you to do so only within an already specified field,

and then only when the description of that field permits you to write there. (You may be able to move the cursor to a position that isn't within any field, but FSCR won't let you write there).

## The keyboard while you're using FSCR

While FSCR has control, most keys have the same meanings as under the Session Manager, but a few are adapted to the different circumstances.

**Keys whose meanings are unchanged:**

Graphics (letters, numbers, blank, etc.)

<backspace> (including <Alt backspace> to form overstrikes)

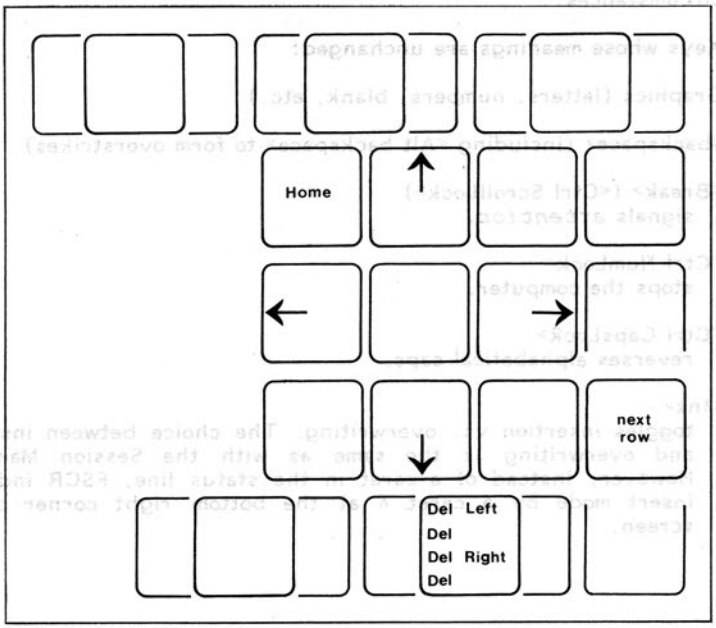
<Break> (<Ctrl ScrollLock>)
signals *attention*.

<Ctrl NumLock>
stops the computer.

<Ctrl CapsLock>
reverses alphabetical caps.

<Ins>
toggles insertion vs. overwriting. The choice between insertion and overwriting is the same as with the Session Manager. However, instead of a caret in the status line, FSCR indicates insert mode by a caret ^ at the bottom right corner of the screen.

**Figure 13-2: FSCR control keypad**



**Keys whose meanings are different in full-screen mode:**

The keys marked ← → ↑ ↓ on the numeric keypad
move the cursor on the screen (and have no effect on the log).

<Home>
moves the cursor to the beginning of the keyable field nearest the top left corner of the screen (that is, the first field you encounter scanning along successive rows from left to right).

<tab>
moves forward along the row you're now on, to the leftmost position in the next keyable field; if it doesn't find another keyable field on this row, it continues on down through successive rows until it finds one.

<shift tab>
moves backward along the row you're now on to the leftmost position within the field. When you're already at the leftmost position, it skips to the leftmost position of the next keyable field to the left on the same row, or, if there isn't a keyable field there, backwards up succeeding rows until it finds one.

<keypad-plus> (the large key at the far right of the numeric pad)
jumps to the leftmost position in the next keyable field lower on the screen than your current position (or to the top of the screen, if you're already at the bottom).

<Del>
deletes the character at the current cursor position.

<Alt Del>
deletes the present character and all characters further to the right on this row of this field.

<Ctrl Del>
deletes the present character and all characters further to the left on this row of this field.

Other keys in the numeric keypad have no function in full-screen mode.

**Function keys.** While FSCR is in control, the ten keys at the left edge of the keyboard do **not** have whatever meanings you may have assigned to them under the Session Manager. Instead, most (but not all--see the next paragraph) pass a "release" signal to FSCR. Since FSCR can report which was pressed, each program that uses FSCR can adopt its own scheme, interpreting each key as a distinct signal. FSCR distinguishes whether you press a function key in base case, or with <shift> or with <Alt> or with <Ctrl>. FSCR uses numbers to identify the keys as follows:

| | |
|---|---|
| 1 through 10 | <F1> through <F10> |
| 11 through 20 | <shift F1> through <shift F10> |
| 21 through 30 | <Alt F1> through <Alt F10> |

**Correspondence between PC keys and 3270 keys.** IBM 3270 display stations come with a wide variety of keyboards. They may have keys labelled <PA1>, <PA2>, and <PA3> (forms of program attention), a key labelled <Clear>, and some number of keys labelled <PF1> up to <PF24>. FSCR encodes the function keys of the PC so that their identifying codes correspond to possible 3270 keys in the following way:

| PC | IBM 3270 |
|---|---|
| <F1> through <F24> | <PF1> through <PF24> |
| <Ctrl F1> through <Ctrl F3> | <PA1> through <PA3> |
| <Ctrl F4> | <Clear> |
| <Ctrl F9> | <Field Mark> |
| <Ctrl F10> | <Dup> |

Note that <PA1>, <PA2>, <PA3>, and <Clear>, and thus the corresponding PC keys <Ctrl F1> through <Ctrl F4> are intended as interrupt signals. They release the screen, but you shouldn't depend on being able to read back what you may have written on the screen before you pressed one of them.

<Dup> and <Field Mark> are displayable characters to which a program may attach special significance, but keying one of them does **not** release the screen. On the 3270, <Field Mark> is displayed as semicolon with macron, but on the PC as a smiling face. On the 3270, <Dup> is displayed as asterisk with macron, but on the PC as something that looks a bit like a door with a knot-hole.

**Status line during full-screen mode.** While FSCR is active, there is no status line. The only status display is the caret superimposed on the last screen position (bottom right corner) when you elect insert mode. The caret does **not** prevent you from using that position as part of a normal field. When you turn off insert, you can see what's actually written at that position. Commands to read the screen always return the real character, even when the caret conceals it.

### Typeahead during full-screen mode

It is possible to type your next entry before the computer has finished executing the preceding one. If the computer is doing something that takes a long time and you type fast, you may be able to get several entries ahead. Use of this feature with the Session Manager is described in chapter 3 **Using SHARP APL/PC**.

In full-screen mode, typeahead continues to work in the same way. Regardless of whether the screen is controlled by the Session Manager or by FSCR, a record of your keystrokes is held in the input buffer. When keyboard input is required, the computer reads the keystrokes from that buffer. The record of your keystrokes is interpreted according to whatever situation prevails at the time input is needed. The same buffered record of keystrokes supplies input to whoever needs it next: the Session Manager or FSCR. This means that it is theoretically possible--but in practice pretty difficult--to key entries that will become the text of fields on the screen even before the screen is visible.

## THE VARIABLES THAT LINK YOUR APL WORKSPACE TO FSCR

Your link to FSCR is a variable that you share with processor 124. Or it may be a pair of variables, used together as a single interface. (The choice between a one-variable and a two-variable interface is explained in the next section.)

On the PC, you can have only one interface to FSCR. (In SHARP APL/370, by sharing several different sets of variables with AP124, you could establish several different screens, and display one or another depending on which of the shared variables you use to request a display.)

The principal variable you offer to 124 must have a name that begins with the letters *ctl* and has no more than 8 letters altogether. (You may also offer a **surrogate name**, so the name appears to you as something else, but appears to AP124 in a form acceptable to it. Surrogate names are discussed in chapter 9.)

### One-variable and two-variable interfaces

You have to communicate to FSCR two different kinds of things: the **command** indicating the action you want it to take, and the **data** for the command. When the shared variable interface was introduced in APL systems (around 1973) APL lacked a convenient way to join objects of different type or shape. Therefore, many auxiliary processors were designed to use **two** shared variables, one to pass commands and another to pass data. In order to make clear which was which, by convention the control variable had a name starting with *ctl* and the data variable a name starting with *dat*. Since you might have several variables shared with the same processor, the convention was that a particular pair must have names that start with *ctl* and *dat* respectively, and have the **same** suffix thereafter; for example, one pair might be *ctlx* and *datx*, another pair might be *ctlwork* and *datwork*, and so on; plain *ctl* and *dat* (with no suffix) are also an acceptable pair.

More recently, with the introduction of enclosed arrays, the two-variable convention has become unnecessary. It's now a simple matter to link together the control information and the data information in a single two-element vector. Moreover, that makes it possible to link multiple commands in a single array so that, setting the shared variable just once, you pass it a sequence of several different commands.

FSCR supports both ways of doing things: with a pair of variables whose names begin with *ctl* and *dat* respectively, or with a single variable whose name begins with *ctl*.

### How FSCR knows whether you're using one shared variable or two

Any time you set the control variable to an **enclosed** array, FSCR assumes that you are using a single-variable interface, and sends its response in the same form.

Any time you set the control variable to a **simple** array, FSCR assumes you are using two variables, and sends its response in the same form.

### Choosing a one-variable or a two-variable interface

The single-variable method, using enclosed arrays, is more modern and conceptually simpler, and is used in the sample function *smpl* (listed at the end of the chapter). The ability to chain multiple commands (possible only with enclosed arrays) may simplify writing, and may be an advantage when you're working in a timeshared system, such as SHARP APL/370. In a timeshared system, your APL workspace and the AP are scheduled quite independently; in that context, there may be a performance advantage to seizing the opportunity to send the AP a whole list of commands at once. On the PC, that advantage doesn't apply. Moreover, on the PC, by using two variables you may avoid a certain amount of processing time involved in enclosing and disclosing the data that's passed back and forth. For this reason, the distributed workspace *fscr* uses two variables, which it calls *ctlS* and *datS*. (The suffix *'S'* indicates that these shared variables are for the screen.)

The descriptions of the various commands later in this chapter show both forms. Note that it's valid to switch back and forth between the two forms in a single application--but you may find it confusing to do so.

### Establishing contact

In what follows, assume that the control variable is called *ctlS* and the data variable (where used) is called *datS*.

You offer processor 124 to share two variables:

```
      124 ⎕svo 2 5⍴'ctlS datS '
1 2
```

Result is at least 1 for each variable, indicating you've made two offers that haven't yet been accepted. (FSCR on the PC manages to make its counter offer for the *dat* variable at once, so you get a 2 at once.)

```
      1 ⎕svc 'ctlS'
1 1 1 1
```

You interlock the control variable but **not** the data variable. Result shows full interlock.

Interlocking the shared variable is not essential on the PC since FSCR will do it anyway. However, it **is** important that *ctlS* be interlocked **before** you make any use of it. On a timeshared system, although AP124 will set the interlock, because the AP is scheduled independently of your workspace, it may not yet have had time to do so when your program tries to set the shared variable. Therefore, your program should **always** set the interlock as soon as you've made your offer--especially if you may use the same program

on a timeshared system.

```
      ⎕svo 2 5⍴'ctlS datS '
2 2
```

124 has accepted the offers to share *ctlS* and *datS*. It may not need *datS* right away--maybe never, if you stick to the single-variable enclosed commands.

It's prudent to verify that sharing has been established before you proceed with sending commands to FSCR. You can verify that the variable is shared by a test such as `→(2=⎕svo 'ctlS')/ok`. (You can see such a test in the function *startfscr*, listed in the notes at the end of this chapter.)

### Single-variable method

The sequence (once you've shared *ctlS* with FSCR) is as follows:

- You set *ctlS* to be a vector each of whose elements is formed by enclosing a command code or the data that the command needs. (When you transmit only one command and it doesn't require data, *ctlS* may be either a one-element vector or a scalar, but in either case containing the enclosed command code.) Suppose *c1* contains your first command, and *d1* is the data it needs; suppose *c2* is your second command and doesn't require data; suppose *c3* is your third command, and *d3* is its data. Then you transmit all three commands to FSCR by:

  ```
  ctlS←c1 ⊃ d1 ⊃ c2 ⊃ c3 ⊃ d3
  ```

- FSCR discloses the first element of *ctlS* and interprets the command it finds there. If that command calls for data, FSCR discloses the next element of *ctlS* to get the data.

- If your *ctlS* has still more elements, FSCR discloses the next one to find the next command, and if need be the one following to find its accompanying data. FSCR continues in this fashion until it has evaluated all the commands in *ctlS*.

- When FSCR has completed work on all the commands contained in *ctlS* (or has given up because it was unable to execute a command), it sets *ctlS* to be a vector of enclosed arrays. It sets the first element to be the enclose of a two-element vector. When all the commands were executed successfully, that first element consists of:

  ```
  <0,n
  ```

  where **n** is the number of commands that FSCR executed.

  When FSCR was unable to complete work on all the commands in *ctlS*, it sets the first element of *ctlS* to consist of:

  ```
  <tr, n
  ```

where **tr** is a code describing the trouble FSCR experienced, and **n** is the (1-origin) number of the command on which FSCR was working when it was unable to continue.

- Your program uses the value of *ctlS*, and stores it under some other name. This is important because the interlocks mean you can't refer directly to *ctlS* again until you've set it (presumably to convey your next command). Since you'll need to refer separately to the various elements of *ctlS*, you need to have *ctlS*'s value available to you without having to use the name '*ctlS*' again. In the sample function *dom* ("do multiple") shown below, you'll see that the value of the shared variable is assigned to *r*, so that subsequent references are to *r* rather than to the shared variable.

- Your program checks the first element of the value returned via *ctlS* to find out whether your command sequence has been carried out.

- The succeeding components in the value returned via *ctlS* contain the various result arrays, one for each command that returns a result. (It's up to you to keep track of which commands called for results, and hence which component contains which result.)

Whenever you have set *ctlS*, you should use it in order to see how things turned out. In the definition that follows, to make sure that set and use don't get out of step, the two are combined in a single statement.

**Sample program to execute multiple FSCR commands:**

```
     ∇ r←dom cmd
[1]    r←ctlS ⊣ ctlS←⊃cmd
[2]   →(×''⍴⍴r)↓End                    ⍝Unless error report
[3]    (errormsg>''⍴r) ⎕signal 124 ⍝Interpret error code
[4]   End:r←1↓r                        ⍝What's left is data
     ∇
```

**Two-variable method**

Once you've offered *ctlS* and *datS*, and sharing of *ctlS* has been established, the sequence is as follows:

- You set *datS* so that it contains whatever data FSCR will need.

- You set *ctlS* to indicate the command you want FSCR to execute. (You must have set *datS* first so that it already contains the data when FSCR receives the command.)

- FSCR interprets the command it finds in *ctlS*. If the command calls for data, FSCR uses *datS*, finding the value you've placed there earlier.

- When the command that FSCR is executing calls for a data result, FSCR stores the result in *datS*.

- When FSCR has completed work on the command, it sets *ctlS*. If work was successful, it writes a zero, which is the code for "success". Otherwise, it writes a code indicating the nature of the problem it encountered.

- Your program looks at *ctlS* to see whether the command has been carried out. Since a set and a use of *ctlS* must occur in pairs (and because it's important not to get them out of step, for example while debugging) the function *do*, shown on the next page, combines set and use in a single statement. (You don't **have** to combine set and use that way--in fact in some situations it's really useful to have your program go off and do something else before checking the return from the AP--but usually it is effective and convenient.)

- If the command has been successful and calls for a data result, your program uses *datS* to find the data that FSCR has put there.

Whenever you have set *ctlS*, you should use it in order to see how things turned out. It is important to do that **before** you set *ctlS* with a new command (unless you're deliberately trying to interrupt FSCR). It is also important to use *ctlS* **before** you attempt to make use of the result that you expect FSCR to put into *datS*.

**Sample program to execute a single FSCR command using a pair of shared variables**. The following defined function passes one command to FSCR and returns its result:

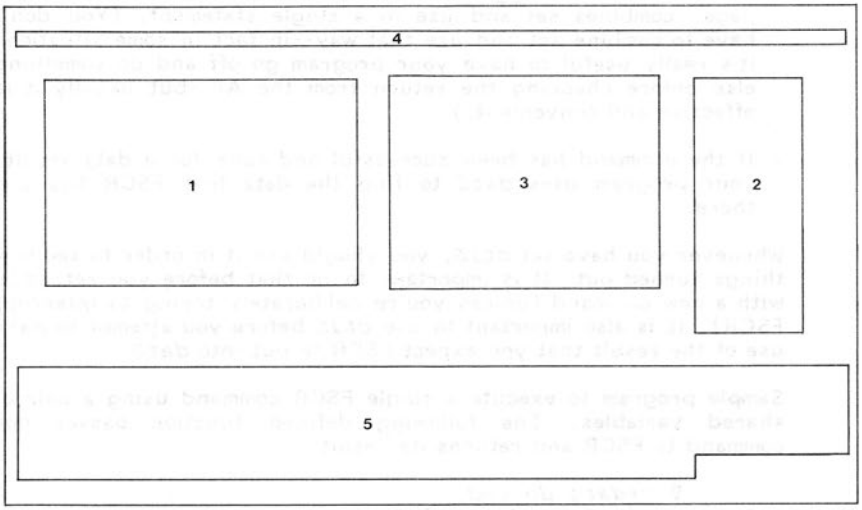
```
      ∇ r←data do cmd
[1]     →(0=⎕nc 'data')/Xfr
[2]      datS←data
[3]  Xfr: r←ctlS ⊣ ctlS←cmd
[4]     →(×''⍴r)⍎Test
[5]      (errormsg r) ⎕signal 124
[6] Test:→(⎕svs 'datS')[⎕io+1]/Get
[7]      r←⍳0
[8]     →0
[9]  Get: r←datS
        ∇
```

## THE GEOGRAPHY OF FIELDS

You can think of the PC screen as a 25×80 matrix. (Or a 25×40 matrix when you're using a colour monitor; FSCR selects the appropriate screen size.) Within the screen area, each field is a rectangle, described by four numbers: the coordinates of its top left corner (row and column, in 1-origin), and its shape (rows and columns).

The figure shows a screen with five fields:

**Figure 13-3: Screen with five fields**




Across the top, a one-row field occupies the entire width of the screen, perhaps for the entry of commands, or to display a caption for the screen as a whole. Its top corner is at row 1, column 1, and its shape is 1×80, so its shape and position are described by the four numbers 1 1 1 80. Since it happens to be specified by the fourth row of the format matrix that describes the screen, the program refers to it as "field 4" (but you can't tell that from the appearance or behaviour of the display).

Below it appear three rectangular fields, each starting on row 3. Field 1 (so called because it is described by the top row of the format matrix) starts at row 3, column 3 and has 7 rows and 34 columns, so its shape and position are described as 3 3 7 34. Field 2 is over at the right side, and starts at row 3, column 69, and occupies 10 rows and 10 columns. (Notice that a 10×10 field doesn't appear square, because character positions are about twice as high as they are wide.) Field 3's position and shape are described by 3 39 7 30.

Field 5 occupies the full width of several rows at the bottom of the screen. It starts at 18 1. However, its description, instead of calling it a 8×80 matrix, describes it as having 1 row and 630 columns. (That's why it ends 10 positions before the bottom corner

of the screen.) FSCR permits a field to be established that way when you describe it as having only one row, with folds imposed where it reaches an edge of the screen. Doing so has some useful consequences that are described in the section **Folded-vector fields**.

### Adjacent fields on the 3270

Terminals of the IBM 3270 family require that some of the space otherwise devoted to storing characters for the screen be allocated instead to attributes of each row of each field. In practice, this means that one or two positions on each row of each field are unavailable for entry or display, so you can't place a field so that a row is adjacent (horizontally) to a row of another field. You have to leave at least one unused position between them. (This prohibition also applies to positions that straddle an edge of the screen; when one field uses the rightmost position on a row, the leftmost position of the next row can't be in a different field. Moreoever, the screen positions wrap around so that the bottom right corner and the top left corner cannot belong to different fields.)

This restriction derives from the IBM 3270 hardware and not from AP124 itself. There is no such restriction when you use FSCR on the PC. However, if your program might also be used on a 3270 terminal, you must bear the restriction in mind, and refrain from laying out fields so that they touch horizontally. The examples shown here are consistent with that restriction.

### Row-structure of a field

When a field has more than one row, in some ways it acts as a collection of separate rows, and in others it acts as a single vector, reshaped to fit into the rectangular space.

When you write data to a field, you supply a vector of characters. Even if you have a matrix of characters whose shape exactly matches the shape of the field in which you want to display it, you still send FSCR the ravel of that matrix. For example, if a field is defined to have 3 rows and 20 columns, to fill it you send FSCR a 60-character vector. FSCR looks up its record of the field's shape, and fills the positions within the field from the data you've supplied. Suppose you call the shape of a field *fs*, and the data to be displayed there *d*. Then FSCR displays the data so that it appears as `fs⍴(×/fs)↑d`.

When you supply more data than will fit in the field, FSCR discards the excess.

When you supply fewer characters than are needed to fill the field, only the characters you supply are displayed. The remaining positions are filled with blanks or with nulls. (Blanks and nulls are discussed a little later in this chapter.)

Similarly, when you read a field from the screen, you receive it as a single-row matrix representing the entire field. If you want to

treat the data from a many-rowed field as a matrix having the same shape that you see on the screen, it's up to you to keep track of the field's shape, and reshape the data you receive accordingly.

For example, suppose you have a 3-by-18 field in which you want to display some names for editing. Suppose the names are *John Paul Jones*, *Maria Casos*, and *Kenichi Nakanishi*. The data you transmit to FSCR must contain a vector including sufficient blanks to arrange each new name at the beginning of a new row:

```
'John Paul Jones    Maria Casos        Kenichi Nakanishi '
```

Although a field is written and read as if it consisted of a single row, there are nevertheless ways in which a field acts as a collection of separate rows. The term **field row** is used to refer to the portion of a field lying on a single row of the screen.

**Overwriting, insertion, and deletion in a field row.**

By default, the keyboard is in no-insert mode. When you move the cursor to a place where a character already appears and there key another character, the new character replaces the first. But when you put the keyboard into insert mode by pressing the <Ins> key, newly keyed characters are inserted ahead of the characters already on the screen; already displayed characters are pushed to the right. However, FSCR lets you do that only until you've filled the field row. Blanks and nulls can be pushed off the end of a field row (and are discarded), but other characters can be deleted only by the <Del> key. (FSCR permits blanks to be pushed off the end of a field row, but 3270 display stations do not; see the discussion of blanks and nulls.)

Characters that are displaced can **not** be pushed into the next field, and can **not** be pushed into a lower row of the same field (except as will be noted in **Folded-vector fields**).

The <Del> key can be used to delete characters in a field row. Pressing <Del> in base case deletes a single character. <Alt Del> deletes the character at the cursor position and the remaining characters rightward in the field row. <shift Del> deletes the character at the cursor position and the remaining characters leftward in the field row.

When you delete, the remaining characters further to the right in the same field row move leftward to fill the vacated positions. But characters are not pulled backward from beyond the boundaries of the individual field row. FSCR fills positions vacated at the end of a field row with nulls. In a full-screen display, nulls (`⎕av[254]`) look just like blanks, but under some conditions they can be distinguished and have different properties. (However, by the time your program reads the field, nulls that were on the screen temporarily may have been converted to blanks; see **Blanks and nulls at the end of a field row.**)

Suppose you wish to edit the names displayed in the 3-by-18 field described earlier. When displayed, they appear on three rows, thus:

*John Paul Jones*
*Maria Casos*
*Kenichi Nakanishi*

Suppose now that you were to edit the display to make the following changes:

- Correct the misspelling of *Casos* to *Casas*. To do that, having used the <Ins> key as necessary to set for no insertion, use the ← → ↑ ↓ controls to position the cursor at the erroneous *o* and there press *a* to replace it with the correct character.

- Replace *Jones'* middle name by his initial. You could do that by positioning the cursor at the *a* in *Paul* and pressing <Del> three times.

- Change *Kenichi Nakanishi* to *Ken Nakanishi, Jr.* Here you'd have to delete some characters and insert at a different place. In this example, if you were to try to insert *, Jr* before shortening the first name, you'd find that there isn't sufficient room remaining in the field row to contain four new characters. However, if you first delete *ichi,* that would make the needed room.

The preceding example serves to introduce the way FSCR treats the space at the end of a field row, discussed in the next section.

### Blanks and nulls at the end of a field row

When you delete characters in a field row (for example, the characters *aul* from within the name *John Paul Jones*), the characters further to the right move leftward to close up the gap. What takes their place at the end of the field row? FSCR always fills the space vacated at the end of a field row with the character **full screen null**, also called FS-null or null. This character displays like a blank, but the computer can distinguish it from the blank you get by pressing the space bar. To make the difference between nulls and blanks visible, suppose we here show an ordinary blank as _ and a null as °. Then the deletion of three characters changes *Jones'* field row from

*John_Paul_Jones___* to *John_P_Jones___*°°°

When FSCR transfers data from your workspace to its storage buffers for display, you may specify that it automatically replace the trailing blanks at the end of each field row with nulls. Whether FSCR does so is controlled by the **blank processing** attribute which is part of each field's description. When the field is initially displayed, the right end of the field row shows:

| With trailing blanks: | With trailing nulls: |
|---|---|
| *John_Paul_Jones___* | *John_Paul_Jones°°°* |

After you've deleted three characters from the middle, the field row contains:

| | |
|---|---|
| *John_P_Jones___°°°* | *John_P_Jones°°°°°°* |

Then when you read from the screen and return the field data to the workspace, if FSCR has been instructed to retain trailing nulls for this field, that portion of the result vector appears exactly as it did on the screen (preceding illustration). But if you elect to have nulls on the screen converted to blanks, the result returned to the workspace is:

| | |
|---|---|
| *John_P_Jones_____* | *John_P_Jones_____* |

### Effect of trailing nulls on insertions

Trailing nulls are a vital concern for programs written for IBM 3270 display stations. That's because 3270s do not permit trailing blanks at the end of a field row to be pushed out by insertions (whereas the PC does). When:

- Your program may also run under timeshared AP124

  and

- You wish to permit insertions into a field row which has space at the right ends of its rows

  then

- It's essential to specify that the trailing blanks on each row be displayed as nulls.

Trailing nulls are of less importance on the PC, but there are some performance advantages in using them in the same way as is required by 3270s.

When you read back the contents of a field, FSCR gives you a similar option to have nulls at the end of each field row converted to blanks. As displayed, a field may have some trailing nulls, either because FSCR wrote them there when it first transferred the data from the workspace, or because you deleted characters and didn't replace them. Those trailing nulls may or may not be important to your application. If you need to make an explicit check for net deletions, you should elect to have FSCR return the nulls unchanged. But if nulls are of no interest, you can direct FSCR to replace them by blanks in the data that it delivers in response to any of the various Read commands.

### The blank-processing attribute of a field

One of the attributes of a field (controlled by the value in column 11 of the format matrix) instructs FSCR how you want trailing blanks handled. The four options are encoded by the numbers 0, 1, 2, and 3, with the following meanings:

| | | When you write data, represent trailing blanks on each field row by: | |
|---|---|---|---|
| | | Blanks | Nulls |
| When you read data, represent nulls by: | Blanks | 0 | 1 |
| | Nulls | 2 | 3 |

The character FS-null occurs in `⎕av` at position 254 (0-origin). You'll need to be able to test for it if you ask FSCR to return nulls. (Don't confuse FS-null with `⎕av[⎕io+0]`, which is also called null; FSCR displays `⎕av[⎕io+0]` in the same way as it displays any other unrecognized characters: as a squish-quad.)

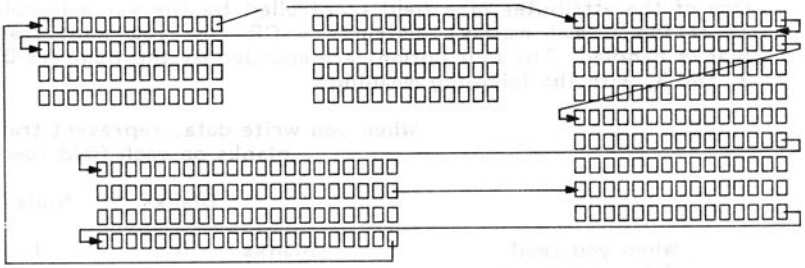
### Cursor skip at the end of a field row

One of a field's attributes established in the format matrix is where the cursor is allowed to go when it reaches the end of a field row. (Setting and changing this attribute is described in the section on field formats.) When the skip attribute is **autoskip**, FSCR disregards positions which are in display-only fields, or in no field at all, and moves immediately to the next keyable field row.

When the cursor reaches the end of a field row, FSCR moves the cursor to the keyable position that is **next on the screen**. If there's an available position in another field and it is reachable by going rightward on the same row, that's where the cursor goes. It doesn't return to the field it was in until it comes down to the beginning of the next row and, moving from left to right in order, again encounters that field.

FSCR calculates which position comes next by considering them in row-major order. That's the order in which you read a book such as this: left to right along each row, then to the beginning of the row below, and so on. If FSCR scans to the end of the screen without finding an keyable field row, it wraps around and continues its search from the top left corner.

**Figure 13-4: Skipping fields**



When the format matrix specifies **noskip**, FSCR advances the cursor to the next position on the screen, regardless of what field it's in or whether or not you're permitted to key there.

### Folded-vector fields

It is possible to arrange a field so that it occupies several rows on the screen, but treats the entire space as a single continuous vector. If you set up a field that way, insertions at the top can push characters from the end of one row into the next; if need be, they can ripple down through the entire field. Similarly, a deletion near the top pulls back all the characters that follow it, all the way to the end of the field, even if that is many rows away.

You can get a field to behave in that way by declaring that it has only one row, as long as the total length of the field. For example, to set up a folded vector field that occupies eight consecutive rows of the screen, you declare that its shape is 1×640. In that case, it extends for 640 consecutive positions, starting from wherever you designate. (It doesn't have to start or end at the edge the screen, and its length doesn't have to be an exact multiple of 80.) The folds in such a field are those forced by the edges of the screen; you can't fold it anywhere else. Thus (except for its first and last rows, which aren't required to fill a full row) it must extend the full width of the screen.

Fields of this sort are useful for free-form information; programs and poems tend to be organized into lines, but prose text is probably more easily handled in large vector fields than in lines.

Although insertions in a folded-vector field are treated in exactly the same way on the PC and a 3270 display station, deletions are not. On both devices, insertions can push down text through the full length of a folded vector. On the PC, deletions can similarly pull back characters throughout the entire length. But on a 3270, deletions are closed up only by moving characters further to the right on the same physical row as the deletion. This behaviour of 3270 terminals arises from their hardware, and is not a design feature of AP124. (Partly in compensation for the nulls thus introduced in the middle of a vector field, a 3270 does not report

them when a program requests AP124 to read the screen. Perhaps that's poetic justice. But it's confusing to the person watching the screen, who finds that visible gaps on the 3270 screen have disappeared from the data returned by AP124 (and have migrated to the end of each row). If your program will be run on both devices, you should be aware of these differences in their behaviour.

## THE FORMAT MATRIX

FSCR describes the format of its fields by an 11-column matrix having one row for each field. When you set or reset the format matrix, you can supply a matrix having 4, 5, 6, or all 11 columns. If you elect a smaller matrix, you get the default values for the remaining columns. The eleven columns can be grouped as follows:

1-4. **Position and shape.** A field is described by the 1-origin coordinates of its first (that is, upper left) position, and its shape (rows and columns). You **must** explicitly specify these four columns for any field you define.

5. **Type.** Whether a field is for display only, or will also permit entries from the keyboard, encoded as follows:

- 0 Keyable (or "unprotected").
- 1 Keyable, numeric only. That is, in addition to blank, the only characters you can key are ¯-+.0123456789). Notice that setting this attribute doesn't cause FSCR to return numeric data, but only to restrict input to numeric characters; your program has to decide what numbers those characters represent.
- 2 Display only (or "protected"). This is the default.

6. **Intensity.** Instead of the normal intensity, you may elect to have the characters in a field displayed with high intensity, or not displayed at all. An undisplayed field is useful for passwords or similar confidential information. A Read command will still report what you key there, even though nothing is visible on the screen. The encodings are:

- 0 Invisible.
- 1 Normal (default).
- 2 High intensity.

(High intensity looks white on a 3270 colour display, and tends to make colours look washed out on the PC colour monitor.)

7. **Colour**. This attribute governs all characters displayed in a field (except as that may be overridden for individual characters by the "write character attributes" command). On a PC without colour, the colour information is ignored. The possible colours are:

0 Green (default).
1 Blue.
2 Red.
3 Pink (i.e. 1+2).
4 Green.
5 Turquoise (i.e. 1+4).
6 Yellow (i.e. 2+4).
7 White (i.e. 1+2+4).

8. **Program symbol set.** Some 3270 display stations permit the program to supply an alternate set of glyphs, called a program symbol set, in substitution for the regular character set. However,

this feature isn't supported for the PC. For compatibility with AP124, this field exists, but on the PC its value must be 0.

9. **Highlighting.** This attribute governs all characters displayed in a field (except as that may be overridden for individual characters by the Reset Character Attributes command). The values are:

0 None (the default).
1 Blink.
2 Reverse video.
4 Underline.

Note: Using the command to set character attributes, you can select a different positive highlighting for individual characters within a field, but you can't set no highlighting for characters when the field itself has positive highlighting.)

10. **Skip.** Where the cursor moves next after reaching the last character of a field row.

0 Skip to beginning of next keyable field row (default).
1 Advance to next physical position (regardless of field).

11. **Trailing characters.** How FSCR displays unused positions at the end of a field row, and how it reads back nulls.

| | **Display trailing blanks as:** | **Read trailing nulls as:** |
|---|---|---|
| 0 | Blanks | Blanks |
| 1 | Nulls | Blanks (default) |
| 2 | Blanks | Nulls |
| 3 | Nulls | Nulls |

**Summary table: columns of the format matrix**

[Default values are in **bold**.]

| 1-2 | 3-4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|
| Top left | Shape | Type | Intensity | Colour | PS | Highlight | Field end | Trailing Write-Read |
| | | 0:I/O<br>1:Num<br>**2:Disp.** | 0:Invsbl<br>**1:Normal**<br>2:High | **0:Green**<br>1:Blue<br>2:Red<br>3:Pink<br>4:Green<br>5:Turquoise<br>6:Yellow<br>7:White | **0** | **0**<br>1:Blink<br>2:Reverse<br>3:Underline | **0:Skip**<br>1:Don't | 0:B : B<br>**1:N : B**<br>2:B : N<br>3:N : N |

## FSCR COMMANDS

### Command 1: Set field formats

This command can be used either to set **all** field formats, or to reset the format of selected previously defined fields. To provide a complete new set of formats, the command is 1. To provide new formats for selected fields, the command is a vector consisting of 1 followed by the numbers of the fields to be revised.

#### Command 1: Set new field formats

```
1-variable:  ctlS←1 ⊃ fmt
             r←ctlS          Enclosed return code.

2-variable:  datS←fmt
             ctlS←1
             r←ctlS          Return code.
```

**fmt** is the format matrix, having one row for each field, and 4, 5, 6 or 11 columns. If you supply a matrix in one of the narrower forms, the default values are assumed.

#### Command 1: Reformat selected fields

```
1-variable:  ctlS←(1,n) ⊃ fmt
             r←ctlS          Enclosed return code.

2-variable:  datS←fmt
             ctlS←1,n
             r←ctlS          Return code.
```

**n** is a vector of the numbers of the previously defined fields whose formats are to be changed.

**fmt** is a format matrix in the same form as for a new format matrix.

Changing the format of a field has no effect on the data available for display in it. If you change a field's format by allowing it less space, FSCR displays as much data as will fit, but the remainder is not discarded from FSCR's buffers (and will again become visible if you subsequently change the format to provide sufficient space to contain it).

### Command 9: Read the current format matrix

```
1-variable:  ctlS←<9
             r←ctlS          Enclosed return code, catenated to
                             enclosed format matrix.

2-variable:  ctlS←9
             r←ctlS          Return code.
             r←datS          Format matrix.
```

Returns the current format matrix, with one row for each field that has been defined.

### Commands 6, 7, 16, 24, and 26: Reset field attributes

Command 6, 7, 16, 24, or 26 resets the values of a particular attribute for selected previously defined fields. Each has the form:

```
1-variable:  ctlS←(c, n) ⊃ a
             r←ctlS                Enclosed return code.

2-variable:  datS←a
             ctlS←c, a
             r←ctlS                Return code.
```

**c** is a command number, as follows:

6 Reset field type.

7 Reset field intensity.

24 Reset field colour.

26 Reset field highlighting.

16 Reset field skip (~2|*a*) and trailing blank treatment (⌊*a*÷2):

| a | Skip | Write trailing | Read nulls |
|---|---|---|---|
| 0 | No-auto | Blanks | Blanks |
| 1 | Auto | Blanks | Blanks |
| 2 | No-auto | Nulls | Blanks |
| 3 | auto | Nulls | Blanks |
| 4 | No-auto | Blanks | Nulls |
| 5 | Auto | Blanks | Nulls |
| 6 | No-auto | Nulls | Nulls |
| 7 | Auto | Nulls | Nulls |

**n** is a vector of field numbers;

**a** is a vector of the same length as **n**, containing the new values for the attribute.

### Commands 2 and 4: Transmit data to be displayed

Two commands permit you to specify the characters to be displayed in selected fields:

2 Respecify characters in FSCR's buffers for the fields indicated and generate a new display of all fields.

4 Respecify characters in FSCR's buffers for the fields indicated (but do not yet generate a new screen display).

1-variable:
```
ctlS←(c,n) ⊃ dmat
r←ctlS              Enclosed return code.
```

2-variable:
```
datS←dmat
ctlS←c,n
r←ctlS              Return code.
```

**c** is the command number, 2 or 4;

**n** is a vector of field numbers;

**dmat** is a matrix of data to be displayed, having one row for each of the fields identified in **n**, and sufficient columns to contain the data of the largest field. Note that a row of **dmat** contains the ravel of the data to be displayed; the division of a field into rows and columns is not apparent in the data provided in **dmat**. (A row containing data for a small field must nevertheless have the same length as all the other rows transmitted together in **dmat**. However, FSCR ignores the positions unneeded for the particular field, so it doesn't matter how the row is padded.)

### Command 12: Set cursor position

1-variable:
```
ctlS←12 ⊃ fld,row,col
r←ctlS              Enclosed return code.
```

2-variable:
```
datS←fld,row,col
ctlS←12
r←ctlS              Return code.
```

Positions the cursor at the indicated position at the next Write/Wait/Read command (command 3). A cursor position is described by three numbers: field number, row, and column (in 1-origin). You can move the cursor to a position that is not within any field by making the first of those numbers 0, in which case the row and column coordinates are the (1-origin) coordinates of the location on the screen as a whole.

### Command 11: Sound the alarm

1-variable:
```
ctlS←<11
r←ctlS              Enclosed return code.
```

2-variable:
```
ctlS←11
r←ctlS              Return code.
```

At the next Write/Wait/Read (command 3), sound the beeper once after displaying the screen and before accepting input from the keyboard.

### Commands 35 and 37: Set character attributes

These two commands permit you to specify how the colours and highlighting of individual character-positions depart from the colour and highlighting specified for the field as a whole:

35 In FSCR's buffers, reset attributes individually for all positions in the indicated field(s).

37 In FSCR's buffers, reset attributes individually for all positions in the indicated fields and generate a new display for all fields.

```
1-variable:  ctlS←(c,n) ⊃ amat
             r←ctlS              Enclosed return code.

2-variable:  datS←amat
             ctlS←c,n
             r←ctlS              Return code.
```

**n** is a vector containing the field numbers of the fields whose attributes are to be reset character by character.

**amat** is a matrix in the same form as for command 4: that is, having one row for each field in **n**, and sufficient columns to provide data for the largest field mentioned in **n**. The data consists of characters, each of which encodes the information regarding colour and highlighting. The colour and highlighting of a character are encoded by

```
⎕av[8 4⊥colour, highlighting]
```

The possible values for **colour** are 0 to 7, and the possible values of **highlight** are 0-3. Values other than zero have the same significance described in the section on the format matrix and field attributes. But 0 means "leave unchanged the attributes already established for the field." (This has the consequence that, with commands 35 and 37, you can turn highlighting **on** but not off; if some colour or highlighting has already been selected for the field as a whole, you can't select "no highlighting" or "no colouring" since in this context 0 is interpreted as "no change in highlighting" or "no change of colour".)

**Caution:** In this command, 1, 2, and 3 select blink, reverse video, and underlining; in the Set formatting and Set highlighting commands, the values 1, 2, and 4 are used.

### Command 3: Write/Wait/Read

```
1-variable:  ctlS←<3
             r←ctlS              Enclosed return code, catenated to
                                 enclosed result.

2-variable:  ctlS←3
             r←ctlS              Return code.
             r←datS              Result vector.
```

Command 3 does three things:

- Generates a new display for all fields (based on the data already in FSCR's buffers).
- Waits for input from the keyboard, permitting editing of any fields except those restricted to display only. Input continues until you release the screen by pressing <return> or any of the function keys to the left of the main keyboard.
- Returns a result indicating how you released the screen, where on the screen the cursor was when you did so, and a list of the fields in which you made any entry or deletion.

The result of command 3 is a vector having at least five elements, and possibly more, as follows:

| 1 | 2 | 3 | 4 | 5 | Remaining elements |
|---|---|---|---|---|---|
| Release code | Modifier | Position of cursor Field | Row | Column | Field numbers of modified fields |
| 0: <return> | 0 | n | n | n | nn... |
| 1: <PF> key | 1-30 | n | n | n | nn... |
| 4: <PA> key | 1-3 | | | | |
| 5: <Clear> | | | | | |

The terms <PA>, <PF> and <Clear> are borrowed from 3270 display stations; on the PC, they refer to uses of the function keys at the left of the keyboard; see the section **Keyboard when you're using FSCR**.

Command 3 does not report directly on the data on the screen, but indicates that the screen has been released, and which fields you changed before you released it. (Commands described in the next section return the actual data from those fields.)

### Commands 5, 30, 31, and 40: Reading what's on the screen

These four commands provide alternative ways of reading the contents of FSCR's buffers (and hence of reading what's on the screen):

5 Return specified fields (as rows of a matrix result).

30 Read fields modified during the last Write/Wait/Read (command 3) and not yet read, as rows of a matrix.

31 Read fields modified during last Write/Wait/Read (command 3), as enclosed vectors.

40 Read entire screen as single matrix.

### Command 5: Read specific fields

| | | |
|---|---|---|
| 1-variable: | *ctlS*←<5,**n** | |
| | *r*←*ctlS* | Enclosed return code, catenated to enclosed result matrix. |
| 2-variable: | *ctlS*←5,**n** | |
| | *r*←*ctlS* | Return code. |
| | *r*←*datS* | Result matrix. |

**n** is vector field numbers.

The result is a matrix, with a row for each element of **n**, and as many columns as necessary to contain the largest field. Rows representing shorter fields are padded with blanks or nulls (as indicated by column 11 of the format matrix).

### Commands 30 and 31: Read modified fields

| | | |
|---|---|---|
| 1-variable: | *ctlS*←<30 | *ctlS*←<31 |
| | *r*←*ctlS* | Enclosed return code, catenated to enclosed result. |
| 2-variable: | *ctlS*←30 | *ctlS*←31 |
| | *r*←*ctlS* | Return code. |
| | *r*←*datS* | Result. |

For either command 30 or command 31, the result contains data from those fields in which you did any keying, insertion or deletion before you released the last screen, and for which you have not yet read the data by an earlier use of command 30 or 31.

The result does not indicate from which fields the data came; you have to keep track yourself of which fields were modified (from the result of command 3).

Command 30 returns a matrix result, with one row for each modified field, and as many columns as needed to contain data from the largest modified field. Rows representing smaller fields are padded with either blanks or nulls, as indicated by the value of column 11 of the format matrix. The data contains no indication of the shape of the field from which it comes; you have to reshape either by noting the shape when you established the field, or by reading the current shape from the format matrix (command 9).

Command 31 returns a vector of enclosed vectors, each of which contains the (ravelled) data from a field that was modified.

**Command 40: Read entire screen**

| | | |
|---|---|---|
| **1-variable:** | *ctlS*←<40 | |
| | *r*←*ctlS* | Enclosed return code, catenated to enclosed result matrix. |
| **2-variable:** | *ctlS*←40 | |
| | *r*←*ctlS* | Return code. |
| | *r*←*datS* | Result matrix. |

The result is a matrix the same shape as the entire screen, containing the characters at each position.

## CHECKLIST: POINTS OF DIFFERENCE BETWEEN FSCR WITH SHARP APL/PC AND AP124 WITH SHARP APL/370

These differences arise principally from differences in the hardware of the PC display and 3270 display stations. For brevity, these are listed here as **PC** and **3270**.

- The 3270 does not permit different fields to occupy horizontally consecutive positions. The PC has no such restriction. A field that ends in column 80 is "horizontally consecutive" with a field that begins in column 1 of the next screen row.

- The PC permits both blanks and nulls to be pushed out of a field row by insertions further to the left. The 3270 permits only nulls to be pushed out in that fashion.

- Where nulls occur in **non**trailing positions in a field row, the result returned to the workspace from the PC shows them at their actual positions, but the result returned by the 3270 compresses them out and shows them bubbled to the right end of each field row.

- Deletions in a folded-vector field on the PC pull back the following characters throughout the entire field (for as many rows as it extends), whereas similar deletions on the 3270 pull back characters only on the same row of the screen (causing the remaining positions on the row to be filled with nulls).

- SHARP APL/370 permits you to share several different sets of variables with AP124 at the same time (each controlling a separate version of the screen). SHARP APL/PC does not permit you to share more than one set of variables with FSCR.

## DISTRIBUTED WORKSPACE *fscr*

Workspace *fscr* contains functions to facilitate work with FSCR. The functions assume that the expression *shareap* 124 has been executed. Those that return an explict result are shown with *r*←. They signal an error if they are unable to complete successfully.

*r*←*shareap*124
Shares variables *ctlS* and *datS* with FSCR. Returns a 1 when coupling is complete.

*format* **fmt**
Command 1: **fmt** is the format matrix containing 4, 5, 6, or 11 columns.

**fld** *reformat* **fmt**
Command 1: Format matrix **fmt** is applied to field numbers **fld**.

*r*←*readformat*
Command 9: The result is the current format matrix.

**fld** *fieldtype* **typ**
Command 6: Type values in field numbers in **fld** are reset to the values in **typ**.

**fld** *intensity* **int**
Command 7: Intensity values in field numbers in **fld** are reset to the values in **int**

**fld** *color* **col**
Command 24: Colour values of field numbers in **fld** are reset to **col**.

**fld** *highlight* **hi**
Command 26: Highlight values for fields in **fld** are reset to **hi**.

**fld** *fieldattr* **att**
Command 16: Attribute values in **att** are applied to field numbers in **fld**.

**fld** *write* **data**
Command 4: Data in a row of character matrix **data** is written to the buffer of the field whose number is in the corresponding position of **fld**.

**fld** *writeattr* **att**
Command 35: Encoded character attributes in a row of **att** are written to the buffer of the field whose number is in the corresponding position in **fld**.

**fld** *immwr* **data**
Command 2: Write data to the buffers of the indicated fields, in the same manner as *write*, and then generate a new display of all fields.

**fld** *immwrattr* **att**
Command 37: Write attributes to the buffers for the indicate fields in the same way as *writeattr* and generate a new display of all fields.

*r←readscreen*
Command 3: Generate a new display for all fields, wait for keyboard input, and report how the screen was released, the position of the cursor at release, and a list of fields that were modified.

*r←getfields* **fld**
Command 5: Data as a character matrix is returned for field numbers **fld**.

*r←readdata*
Command 30: Data as character matrix is returned for fields which were modified during the last *readscreen* operation and have not yet been read by *readdata* or *readdatax*.

*r←readdatax*
Command 31: Same as *readdata* except that instead of returning fields as the successive rows of a matrix, they are returned as a vector of enclosed vectors.

*r←getscreen*
Command 40: A character matrix representing the current characters on the screen is returned.

**fld** *setcursor* **row, col**
Command 12: Sets the position at which the cursor will appear at the next use of *readscreen*.

*alert*
Command 11: Sets a flag so that at the next use of *readscreen* the alarm will be sounded.

## COMMAND SUMMARY

| *ctlS* | *datS* | **Description** |
|---|---|---|
| 0 | * | No operation |
| 1 | fmt matrix | Format screen into fields |
| 1,fld | fmt matrix | Reformat selected fields |
| 2,fld | data | Display data in field immediately |
| 3 | * | Display, wait for input, report |
| 4,fld | data | Supply data for later display |
| 5,fld | * | Read selected fields |
| 6,fld | type | Change field type |
| 7,fld | intensity | Change field intensity |
| 9 | * | Read format matrix |
| 11 | * | Set alarm to sound at next display |
| 12 | position | Set cursor position |
| 16 | attribute | Change field attributes |
| 20 | * | No operation |
| 24,fld | colour | Change field colour |
| 26,fld | highlight | Change field highlighting |
| 30 | * | Get data for modified fields |
| 31 | * | Get data for modified fields in enclosed form |
| 35,fld | attributes | Change character attributes for next display |
| 37,fld | attributes | Change character attributes for current display |
| 40 | * | Read entire screen |

## RETURN CODE SUMMARY

| | |
|---|---|
| 0 | Normal return, operation successful |
| 11 | Control variable rank error |
| 12 | Control variable length error |
| 13 | Control variable domain error |
| 14 | Invalid command |
| 15 | Request to position cursor in undefined field |
| 21 | Data variable rank error |
| 22 | Data variable length error |
| 23 | Data variable length error |
| 24 | Data variable not shared |
| 30 | Invalid field number |
| 32 | Defined field extends beyond the screen |
| 33 | Reference outside field definition |
| 37 | Invalid field type |
| 38 | Invalid field intensity |
| 99 | Hardware error |
| 201 | Invalid colour |
| 202 | Invalid highlight |
| 204 | Invalid program symbol set |
| 205 | Internal buffer overflow |
| 206 | Invalid skip column in format matrix |
| 207 | Invalid nulls column in format matrix |

## DISTRIBUTED WORKSPACE *fscrex*

```
     ⎕lx
'ctlS & datS',((2∨.≠startfscr)/' not'),' shared with fscr'

/     ∇ r←startfscr; nm
[1]    r←124 ⎕svo nm← 2 5 ⍴'ctlS datS '
[2]    r←1 ⎕svc nm
[3]   Test:→(2∧.=r←⎕svo nm)/0
[4]    'waiting for ap 124'
[5]    r←124 ⎕svo nm
[6]    →⎕sc/Test
     ∇
```

*do* and *dom* and *sample* listed in the body of the chapter.

```
   ∇ r←α smpl ω; f1; f2; f3; f4; rho2; rho3; d
 [1] rho2← 15 50 ⌊⍴ω← (¯2↑ 1 1,⍴ω)⍴ω   ⍝size of names matrix
 [2] rho3← 15 30 ⌊⍴α←⍕(2↑(⍴α),1 1)⍴α   ⍝nums as 1-col matrix
 [3] f1←  1  2  1 78, 2 2 0 0, 2 1 0
 [4] f2←  3  3, rho2, 0 1 0 0, 2 0 1   ⍝commas help the eye
 [5] f3←  3  5, rho3, 1 1 0 0, 2 0 2
 [6] f4← 18 30,1 600, 0 1 0 0, 2 1 3
 [7] r←dom 1⊃ 4 11 ⍴f1,f2,f3,f4        ⍝establish formats
 [8] r←dom 2 1⊃' SAMPLE FSCR PROGRAM.
       Experiment: try inserting and
       deleting in each field.'
 [9] r←dom 4 2⊃(,ω)⊃4 3⊃(,α)⊃4 4⊃
       320⍴'This is a free-form
       text field. '
[10] r←dom 11⊃ 12⊃ 2 1 1⊃ 3⊃ 5 2 3     ⍝beep; cursor;
                                        disp/wait; read 2,3
[11]  d←>''⍴⍞r                          ⍝revised flds 2 and 3
[12]  r←r[⎕io], (rho2⍴d[⎕io;]) ⊃ rho3⍴d[⎕io+1;]
     ∇

     ∇ z←α separate ω
[1]    ⍝(0=⎕nc 'α')/'α←'' '''      ⍝default delim is blank
[2]    z←¯1⍤<,α,ω
[3]    z←(⌈/,⍴⍤>z)↑⍤>z
     ∇


     ∇ msg←errormsg ω
[1]    →(2=⎕svo 'ctlS')/OK
[2]    msg←'control variable not shared'
[3]    →0
[4]   OK:msg←(⍕ω),' ',Emsgs[Enos⍳''⍴ω;]
     ∇


       Enos
0 11 12 13 14 15 21 22 23 24 30 32 33 37 38 99 201 202 204
       205 206 207
```

*Emsgs*

```
Success
Control rank error
Control length error
Control domain error
Invalid command
Cursor to undefined field
Data rank error
Data length error
Data domain error
No data variable shared
Invalid field number
Proposed field would be off screen
Cursor would be outside field
Proposed type invalid
Proposed intensity invalid
Hardware error
Proposed colour invalid
Proposed highlight invalid
Proposed PS invalid
Internal buffer overflow
Proposed skip invalid
Proposed null-processing invalid
Return code not recognized
```

## FSCR START PARAMETERS

**Bn**
specifies the size of the buffer used by FSCR, in bytes. This buffer is used for all the storage requirements of FSCR, including the screen image and the shared variables. **n** may be any integer from 1000 to 63530. If you omit B, the buffer has 15000 bytes.

**Fn**
specifies the method used by FSCR in writing to a colour/graphics screen. **n** may be 0, 1, or 2. If you omit F, method 2 is used.

**F0**
indicates the fastest method, but results in disconcerting screen flashes on most monitors.

**F1**
indicates a fancier but slower method that has less noticeable flashes than F0.

**F2**
indicates a method that writes to the screen only during the horizontal flyback and has no flashes, but is slower than F0 or F1. With F2, a fast typist can get ahead of the display.

In most cases, you will want F2. However, if your colour monitor is sophisticated enough to avoid the flashes itself, then you would want to use F0 or F1.

**Mn**
indicates whether FSCR is to use monochrome attributes, even though the device appears to DOS as a colour monitor. **n** may be either 0 (specifying that FSCR is not to use monochrome attributes) or 1. If you omit M, FSCR does not use monochrome attributes for what appears to be a colour monitor. The M parameter may be useful with some special, non-IBM display adapter boards.

**Qn**
indicates whether FSCR is to display its start message. **n** may be either 0 (specifying that the message is not to be displayed) or 1. If you omit Q, the message is displayed.

**Un**
specifies the processor number to be used by FSCR. **n** may be any integer from 2 to 999. If you omit U, FSCR uses processor number 124.

# Appendix A: KEYBOARD AND CHARACTER SET

This appendix shows the actions of the **main keyboard** and **control keypad**, and how the SHARP APL character set is displayed.

The **function keypad**--keys <F1> through <F10>--has no predefined action in SHARP APL/PC. You can, however, define these keys.

For more information about the Session Manager, see chapter 6. For information about how to change the keyboard layout, see appendix B **Customizing the keyboard and character set.**

## MAIN KEYBOARD

Figure A-1 shows the main keyboard in effect when you start SHARP APL/PC. This keyboard is based on the North American layout.

**Base case:** The glyph in the lower left of each keytop shows the character produced when you press the key in base case, that is, when you're not pressing any other key. (When caps reversal is on, you produce base-case letters by pressing a letter key and <shift> at the same time.)

**Alternate case:** The glyph in the upper right of each keytop shows the character produced when you press that key **and** the <Alt> or <CapsLock> key at the same time. (SHARP APL/PC treats the key marked <CapsLock> as the right-hand <Alt> key.) The alternate-case characters may be marked by stickers on the fronts of the keys.

**Shift case:** The glyph in the upper left of each keytop shows the character produced when you press that key **and** a <shift> key at the same time. (When caps reversal is on, you produce shift-case letters by pressing a letter key without also pressing <shift>.)

**Control case:** No graphic keys have any action when you also press the <Ctrl> key.

## Key names

In this handbook, some keys are called by English words rather than by their symbols. Table A-1 gives the key number, North American keytop marking, and handbook name. (Every key has a key number; see figure B-3 in appendix B.)

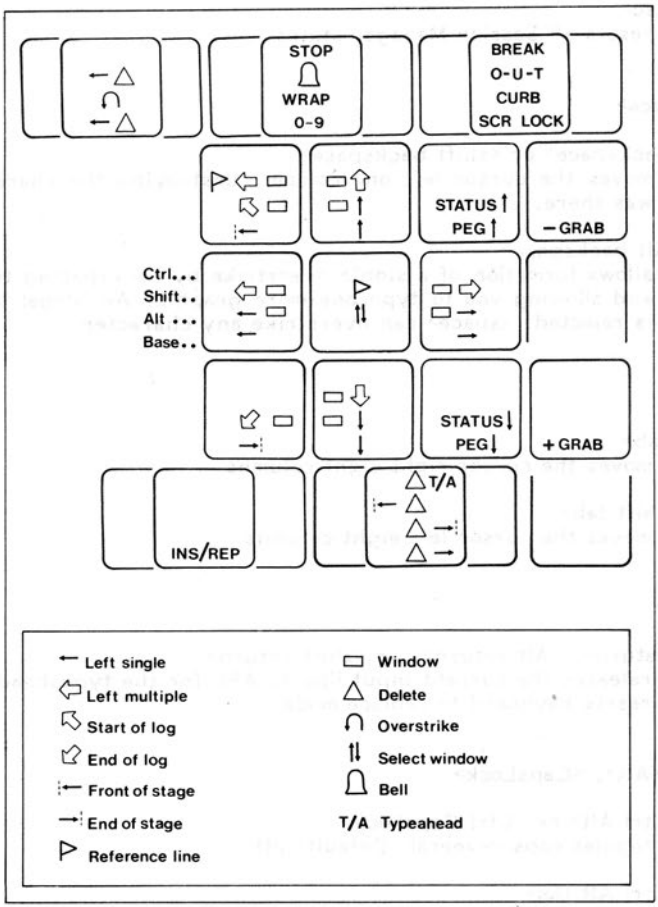
Table A-1

**Key names**

| Key number | Keytop marking | Key name |
|---|---|---|
| 12 | - | minus |
| 13 | = | equals |
| 14 | ← | backspace |
| 15 | →\| | tab |
| 28 | [bent arrow] | return |
| 39 | ; | semicolon |
| 40 | ' | quote |
| 41 | ` | backquote |
| 42 | \ | backslash |
| 51 | , | comma |
| 52 | . | period |
| 53 | / | slash |
| 57 | [space bar] | space |
| **Control keypad** | | |
| 72 | ↑ | up |
| 74 | - | keypad-minus |
| 75 | ← | left |
| 76 | 5 | keypad-5 |
| 77 | → | right |
| 78 | + | keypad-plus |
| 80 | ↓ | down |

**Figure A-1: Main keyboard**

**Figure A-2: Control keypad**



## CONTROL ACTION BY KEY

Keys are given in order by their key numbers. Every key has a unique key number; for all keys' key numbers, see figure B-3 in appendix B.

### <Esc>

<Esc>
resets all Session Manager states.

### <backspace>

<backspace> or <shift backspace>
moves the cursor left one column, destroying the character that was there.

<Alt backspace>
allows formation of a single overstrike by backspacing the cursor and allowing you to type one more graphic. An illegal overstrike is rejected. <space> can overstrike any character.

### <tab>

<tab>
moves the cursor right eight columns.

<shift tab>
moves the cursor left eight columns.

### <return>

<return>, <Alt return>, or <shift return>
releases the current input line to APL (or the typeahead buffer); resets keyboard to replace mode.

### <Ctrl>, <Alt>, <CapsLock>

<Ctrl Alt> or <Ctrl CapsLock>
toggles caps reversal. Default: off.

<Ctrl Alt Del>
restarts DOS. All unsaved work is lost; serious damage may result to untied and open files.

### <PrtSc>

<shift PrtSc>
attempts to print what's shown on the screen.

**<NumLock>**

<NumLock>
activates or deactivates the numeric keypad. When the numeric keypad is activated, "NP" appears in the status line. Default: unactivated.

<Alt NumLock>
enables/disables wrapping of lines in current window. Default: enabled.

<shift NumLock>
enables/disables the keyboard alarm bell. Default: disabled.

<Ctrl NumLock>
causes the entire system, APL, and all APs, to stop until any key (except <NumLock>) is pressed.

**<ScrollLock>**

<ScrollLock>
locks/unlocks scrolling of the lower window. Default: unlocked.

<Alt ScrollLock>
curbs/releases output. Default: no curb requested.

<shift ScrollLock>
signals input interrupt (event 1004) to APL (similar to SHARP APL/370's *o* BACKSPACE *u* BACKSPACE *t*).

<Break> (<Ctrl ScrollLock>)
signals attention (event 1002) to APL. If you press <Break> twice, you may signal a strong interrupt (event 1003) to APL.

**<Home>**

<Home>
moves the cursor to the first column of the input window.

<Alt Home>
moves the current window to show the oldest line in the session log.

<shift Home>
moves the current window to show the reference line.

**<up>**

<up>
moves the cursor up one row.

<Alt up>
moves the current window up one row, that is, toward older lines in the session log.

<shift up>
moves the current window up half its height.

**<PgUp>**

<PgUp>
moves the peg up one line.

<Alt PgUp>
moves the status line up one row, thus closing the upper window by one row.

**<keypad-minus>**

<keypad-minus>
completely replaces whatever's in the input stage with the entire pegged line (including leading blanks).

**<left>**

<left>
moves the cursor left one column.

<Alt left>
moves the current window left one column; no effect in wrap mode.

<shift left>
moves the current window left twenty columns; no effect in wrap mode.

**<keypad-5>**

<Alt keypad-5>
selects the other window. The current window is indicated by an arrow in the status line.

<shift keypad-5>
sets the reference line at the current peg.

**<right>**

<right>
moves the cursor right one column.

<Alt right>
moves the current window right one column; no effect in wrap mode.

<shift right>
moves the current window right twenty columns; no effect in wrap mode.

**<keypad-plus>**

<keypad-plus>
copies the entire pegged line (including any leading blanks) to the cursor (under control of insert/replace mode).

**<End>**

<End>
moves the cursor to the last typed character in the input line.

<shift End>
moves the current window to show the last line in the session log.

**<down>**

<down>
moves the cursor down one row.

<Alt down>
moves the current window down one row, that is, toward newer lines in the session log.

<shift down>
moves the current window down half its height.

**<PgDn>**

<PgDn>
moves the peg down one line.

<Alt PgDn>
moves the status line down one row, thus taking one row from the lower window and giving it to the upper.

**<Ins>**

<Ins>
toggles insert/replace mode. Default: replace mode.

**<Del>**

<Del>
deletes the character at the cursor.

<Alt Del>
deletes the input line from the cursor to the last typed character in the line.

<shift Del>
deletes the input line from the cursor to the beginning of the input stage.

<Ctrl Del>
deletes the typeahead buffer.

## CHARACTER SET

Table A-2 shows the displayable SHARP APL/PC character set, as displayed with the APL character ROM and the IBM factory-equipped character ROM. Also given are the □*av* index, the permitted overstrikes, and the keystrokes that elicit the character.

(The overstrikes that form characters, and the hardware characters used are both determined by Session Manager tables. For information about these tables, see appendix B **Customizing keyboard and character set.**)

**Nondisplayable** characters are shown as squish-quads with the APL ROM, and PL/I NOT symbols with the IBM ROM.

* indicates that the keystroke has effect only when the numeric keypad is activated.

□*av* index is given with origin 0.

Table A-2

SHARP APL/PC character set

| □*av* index | Over-strike | APL | IBM | keystroke |
|---|---|---|---|---|
| 4 | *s*\| | $ | $ | shift 4 |
| 5 | c\| | ¢ | ¢ | |
| 14 | | [ | [ | [ |
| 15 | | ] | ] | ] |
| 16 | | ( | ( | shift 9 |
| 17 | | ) | ) | shift 0 |
| 18 | | ; | ; | semicolon |
| 19 | | / | / | slash |
| 20 | | \ | \ | backslash |
| 21 | | ← | ← | Alt [ |
| 22 | | → | → | Alt ] |
| 25 | | ¨ | ♣ | Alt 1 |
| 26 | | + | + | shift equals |
| 27 | | - | - | minus |
| 28 | | × | Æ | Alt minus |
| 29 | | ÷ | ÷ | Alt equals |
| 30 | | * | * | shift 8; PrtSc; Alt P |
| 31 | | ⌈ | ì | Alt S |
| 32 | | ⌊ | Å | Alt D |
| 33 | | \| | ▮ | Alt M |
| 34 | | ∧ | ∧ | Alt 0; shift 6 |
| 35 | | ∨ | • | Alt 9 |
| 36 | | < | < | shift comma; Alt 3 |
| 37 | | ≤ | ≤ | Alt 4 |
| 38 | | = | = | equals; Alt 5 |
| 39 | | ≥ | ≥ | Alt 6 |
| 40 | | > | > | shift period; Alt 7 |

| □av index | Over-strike | APL | IBM | keystroke |
|---|---|---|---|---|
| 41 | | ≠ | ≠ | Alt 8 |
| 42 | | α | α | Alt A |
| 43 | | ∈ | ∈ | Alt E |
| 44 | | ⍳ | ⍳ | Alt I |
| 45 | | ⍴ | √ | Alt R |
| 46 | | ⍵ | ⍵ | Alt W |
| 47 | | , | , | comma |
| 48 | '. | ! | ! | shift 1 |
| 49 | o\| | ⍱ | ⍱ | |
| 50 | | ⊥ | ∪ | Alt B |
| 51 | | ⊤ | ⊤ | Alt N |
| 52 | | ○ | ∘ | Alt O |
| 53 | | ? | ? | shift slash; Alt Q |
| 54 | | ~ | ~ | Alt T; shift backquote |
| 55 | | ↑ | ↑ | Alt Y |
| 56 | | ↓ | ↓ | Alt U |
| 57 | | ⊂ | ⊂ | Alt Z |
| 58 | | ⊃ | ⊃ | Alt X |
| 59 | | ∩ | ∩ | Alt C |
| 60 | | ∪ | ∩ | Alt V |
| 61 | | - | - | shift minus; Alt F |
| 62 | o\ | ⍲ | ⍲ | |
| 63 | ⊤⊥ | ⌶ | § | |
| 64 | | ⍟ | ⍟ | Alt J |
| 65 | | □ | ò | Alt L |
| 66 | □' | ⍞ | ù | Alt comma |
| 67 | o* | ⍣ | ☼ | |
| 68 | ^~ | ⍱ | Ω | |
| 69 | v~ | ⍲ | σ | |
| 70 | ∩o | ⍀ | a | Alt backslash |
| 71 | Δ\| | ⍋ | ▲ | |
| 72 | ∇\| | ⍒ | ▼ | |
| 73 | o- | ⍜ | Θ | |
| 74 | /- | ⍒ | δ | |
| 75 | \- | ⍋ | o | |
| 76 | □÷ | ⌸ | ÿ | |
| 77 | ⊤o | ⍤ | ƒ | Alt slash |
| 78 | ⊥o | ⍥ | J | Alt period |
| 79 | | ◇ | ♦ | Alt backquote |
| 80 | | ⊣ | ⌊ | Alt quote |
| 81 | | ⊢ | ↔ | Alt semicolon |
| 82 | ]o | } | } | shift ] |
| 83 | [o | { | { | shift [ |
| 86 | | *a* | *a* | A |
| 87 | | *b* | *b* | B |
| 88 | | *c* | *c* | C |
| 89 | | *d* | *d* | D |
| 90 | | *e* | *e* | E |
| 91 | | *f* | *f* | F |
| 92 | | *g* | *g* | G |
| 93 | | *h* | *h* | H |
| 94 | | *i* | *i* | I |

| □av index | Over-strike | APL | IBM | keystroke |
|---|---|---|---|---|
| 95 | | j | j | J |
| 96 | | k | k | K |
| 97 | | l | l | L |
| 98 | | m | m | M |
| 99 | | n | n | N |
| 100 | | o | o | O |
| 101 | | p | p | P |
| 102 | | q | q | Q |
| 103 | | r | r | R |
| 104 | | s | s | S |
| 105 | | t | t | T |
| 106 | | u | u | U |
| 107 | | v | v | V |
| 108 | | w | w | W |
| 109 | | x | x | X |
| 110 | | y | y | Y |
| 111 | | z | z | Z |
| 112 | | Δ | æ | Alt H |
| 113 | a_ | A | A | shift A |
| 114 | b_ | B | B | shift B |
| 115 | c_ | C | C | shift C |
| 116 | d_ | D | D | shift D |
| 117 | e_ | E | E | shift E |
| 118 | f_ | F | F | shift F |
| 119 | g_ | G | G | shift G |
| 120 | h_ | H | H | shift H |
| 121 | i_ | I | I | shift I |
| 122 | j_ | J | J | shift J |
| 123 | k_ | K | K | shift K |
| 124 | l_ | L | L | shift L |
| 125 | m_ | M | M | shift M |
| 126 | n_ | N | N | shift N |
| 127 | o_ | O | O | shift O |
| 128 | p_ | P | P | shift P |
| 129 | q_ | Q | Q | shift Q |
| 130 | r_ | R | R | shift R |
| 131 | s_ | S | S | shift S |
| 132 | t_ | T | T | shift T |
| 133 | u_ | U | U | shift U |
| 134 | v_ | V | V | shift V |
| 135 | w_ | W | W | shift W |
| 136 | x_ | X | X | shift X |
| 137 | y_ | Y | Y | shift Y |
| 138 | z_ | Z | Z | shift Z |
| 139 | Δ_ | ⍙ | ± | |
| 140 | | 0 | 0 | 0; keypad-0* |
| 141 | | 1 | 1 | 1; keypad-1* |
| 142 | | 2 | 2 | 2; keypad-2* |
| 143 | | 3 | 3 | 3; keypad-3* |
| 144 | | 4 | 4 | 4; keypad-4* |
| 145 | | 5 | 5 | 5; keypad-5* |
| 146 | | 6 | 6 | 6; keypad-6* |
| 147 | | 7 | 7 | 7; keypad-7* |

| □*av* index | Over-strike | APL | IBM | keystroke |
|---|---|---|---|---|
| 148 | | 8 | 8 | 8; keypad-8* |
| 149 | | 9 | 9 | 9; keypad-9* |
| 150 | | . | . | period |
| 151 | | ¯ | - | Alt 1; keypad-minus* |
| 152 | | BLANK | | space; keypad-plus* |
| 153 | | ' | ' | quote, Alt K |
| 154 | | : | : | shift semicolon |
| 155 | | ∇ | ∇ | Alt G |
| 156 | | | | NEWLINE |
| 158 | | | | BACKSPACE |
| 159 | | | | LINEFEED |
| 160 | ∇~ | ⍢ | !! | |
| 163 | =_ | ≡ | ≡ | |
| 164 | ∘¨ | ⍤ | π | |
| 165 | O¨ | ⍬ | Σ | |
| 166 | | *a* | *a* | |
| 167 | | *b* | *b* | |
| 168 | | *c* | *c* | |
| 169 | | *d* | *d* | |
| 170 | | *e* | *e* | |
| 171 | | *f* | *f* | |
| 172 | | *g* | *g* | |
| 173 | | *h* | *h* | |
| 174 | | *i* | *i* | |
| 175 | | *j* | *j* | |
| 176 | | *k* | *k* | |
| 177 | | *l* | *l* | |
| 178 | | *m* | *m* | |
| 179 | | *n* | *n* | |
| 180 | | *o* | *o* | |
| 181 | | *p* | *p* | |
| 182 | | *q* | *q* | |
| 183 | | *r* | *r* | |
| 184 | | *s* | *s* | |
| 185 | | *t* | *t* | |
| 186 | | *u* | *u* | |
| 187 | | *v* | *v* | |
| 188 | | *w* | *w* | |
| 189 | | *x* | *x* | |
| 190 | | *y* | *y* | |
| 191 | | *z* | *z* | |
| 197 | | ⍀ | \ | shift backslash |
| 198 | | ` | ` | backquote |
| 199 | | # | # | shift 3 |
| 200 | | @ | @ | shift 2 |
| 201 | | " | " | shift quote |
| 206 | | % | % | shift 5 |
| 207 | | & | & | shift 7 |
| 241-254 | | line-drawing characters | | |

# Appendix B: CUSTOMIZING THE KEYBOARD AND CHARACTER SET

The Session Manager, as distributed, is customized for:

- the English language;
- the North American PC keyboard layout; and
- the I.P. Sharp APL character ROM.

You can use the information in this appendix to customize your copy of the Session Manager for:

- languages other than English (national-use characters);
- keyboard layouts other than North American; and
- character ROMs other than I.P. Sharp's.

Origin 0 is used throughout this appendix.

## Session Manager tables

The Session Manager uses four tables to support keyboard interpretation and display:

- hardware character table
- overstrike table
- keystroke table
- function key table

Changing these tables affects the display on the monitor, and the effect of keystrokes.

## Hardware character table

The hardware character table defines which hardware character is used to display each element of ⎕*av*. A **hardware character** is one of the characters defined as an element of the character ROM on your PC's display adapter.

The hardware character table is a simple translate table: it has 256 elements; the value of the *n*th element of the table indicates which ROM element displays the *n*th element of ⎕*av*.

For example, ⎕*av*[86] is the APL letter *a* from the first alphabet. In the distributed hardware character table, the corresponding value is 97, indicating that hardware character (ROM element) 97 is the display for the letter *a*. In figure B-1, which shows the hardware characters defined by the APL character ROM, you'll find that element 97 (97↔1+6×16; that is, row 1, column 6) is *a*.

You can modify the hardware character table to support the national-use characters you require. Consult the tables in the **SHARP APL Pocket Reference** to determine where in ⎕*av* you can place national-use characters.

You can also modify the hardware character table to support a character ROM other than the one available from I.P. Sharp.

You cannot place a character at an "illegal" position in ⎕*av*; consult the Pocket Reference for all legal positions. You also can't redefine the hardware character table to change the display of ⎕*av* elements 0, 1, 156, 158, and 159 (IDLE, NULL, NEWLINE, BACKSPACE, and LINEFEED, respectively).

### Overstrike table

The overstrike table defines which overstrikes are acceptable to APL. The default value of this table is given in appendix A, table A-1.

You can modify the overstrike table to add national-use characters. For example, *a* overstruck by ¨ can be defined to form a national-use character.

### Keystroke table

The keystroke table specifies how the Session Manager interprets your keystrokes. You can modify the keystroke table to support keyboard layouts other than the North American one, and national-use characters not supported in the distributed Session Manager.

You can view the keystroke table as an 83×5 matrix, with one row for each of the PC's 83 keys, and one column for caps reversal, and one column each for the meaning of the keystroke in base, shift, control, and alternate case.

1 in the caps-reversal column indicates that the key is subject to caps reversal; 0 that it isn't. When you get the Session Manager, only the 26 letter keys are subject to caps reversal.

In the other four columns, a positive value represents a ⎕*av* index. 0 indicates that the keystroke has no effect. A negative value represents a Session Manager action. The codes for these actions are given in **Session Manager actions**, below.

Don't redefine these keystrokes:

<Ctrl> in any case (key 29)
<shift> in any case (keys 42 and 54)
<shift PrtSc> (key 55)
<Alt> in any case (key 56)
<CapsLock> in any case (key 58)
<NumLock> or <Ctrl NumLock> (key 69)
<Ctrl ScrollLock> (key 70)

### Function key table

The function key table contains the function key definitions.

**Workspace *sesm***

Distributed workspace *sesm* contains functions you can use to manipulate the Session Manager tables.

Making the changes is trivial. The hard part is deciding which changes to make. Keep it simple. Only make absolutely necessary changes, and remember to keep both eyes on future compatibility: with your own organization, your own country, and the world.

You can use *sesm*'s functions to experiment with changes to the running Session Manager (that is, to change only the copies of the tables in the PC's memory). When you're satisfied with these provisional changes, you can use other functions to effect the changes on the copy of the Session Manager on disk.

After you've changed the copy on disk, APL will henceforth start with the modified tables.

The Session Manager tables are stored in file CP8X.EXE. Before you modify your production copy of CP8X.EXE, make sure that you have a duplicate so that you can easily recover from changes that were in error.

There are six functions for working with the tables. You identify the table you want to work with by specifying its **tablename**:

| **tablename** | **meaning** |
|---|---|
| *rom* | ROM table |
| *over* | overstrike table |
| *keys* | keystroke table |
| *fkeys* | function key table |

*get*

table←*get* tablename

Returns the current definition of the table specified by **tablename**. The result is a two-element vector of enclosures: the first element is the table identifier (the value of **tablename**); and the second element is the table definition.

*mod*

table←table *mod* change

Returns the table in the left argument updated by the change in the right argument. The format and meaning of a change is different for each of the four tables. *mod* applies only one change at a time.

A change to the hardware character table is a two-element integer vector of the ⎕*av* index of the character and the number of the hardware character to be used.

A change to the overstrike table is a three-element integer vector if an overstrike entry is being added or changed. The vector is the *⎕av* indices of the first, second, and composite characters. If an overstrike table change is only two elements, then that overstrike is removed from the table.

A change to the keystroke table is a six-element vector. The first element is the key number (see figure B-3); the second element is a boolean indicating whether the key is subject to caps reversal; and the remaining elements are the *⎕av* indices for base, shift, control, and alternate cases.

An function key table change is a numeric vector as defined in chapter 6 in the discussion of *⎕arbin* support for function keys. It is probably more convenient to use the functions in distributed workspace *fkeys* to set function key definitions.

*set*

*set* table

The *set* function sets the table in its argument. The changes done by *mod* are now in effect.

*getsuper*

table ←'filespec' *getsuper* tablename

The *getsuper* function is similar to the *get* function. The table is read with DOSX from the file specified in **filespec** (usually CP8X.EXE or a copy).

*setsuper*

'filespec' *setsuper* tablename

The *setsuper* function is similar to the *set* function. The table indicated by **tablename** is read from the current Session Manager and is written with DOSX to the file specified in **filespec**. The changes are in effect when the system is next loaded using that file.

*format*

*format* table

Displays **table**.

**Sample session**

```
)load sesm
```

```
rt←get rom
rt←rt mod 203 132      define a national-use character to
                       be ROM element 132 (a overstruck
                       with ¨)
set rt                 make it effective
⎕av                    see the effect
ot←get over
ot←ot mod 25 86 203    adds overstrike a¨
set ot
```

Now try entering the new overstrike.

The French keyboard layout exchanges the North American keyboard's <A> and <Q> keys. This is how you'd change the Session Manager to match:

```
⎕av⍳'aAqQ??'           Displays the ⎕av indices of the three
                       characters keyable from each key.
86 113 42 102 129 53
      kt←get keys
```

The <A> key on the North American layout is key 30, and the <Q> key is key 16. (For key numbers, see figure B-3.)

```
kt←kt mod 16 86 113 42
kt←kt mod 30 102 129 53
set kt
```

Now experiment with keys 16 and 30.

Figure B-4 is a blank picture of the keyboard layout that you can use to record your custom keyboard layout. Feel free to photcopy it.

**Figure B-1: APL character ROM**

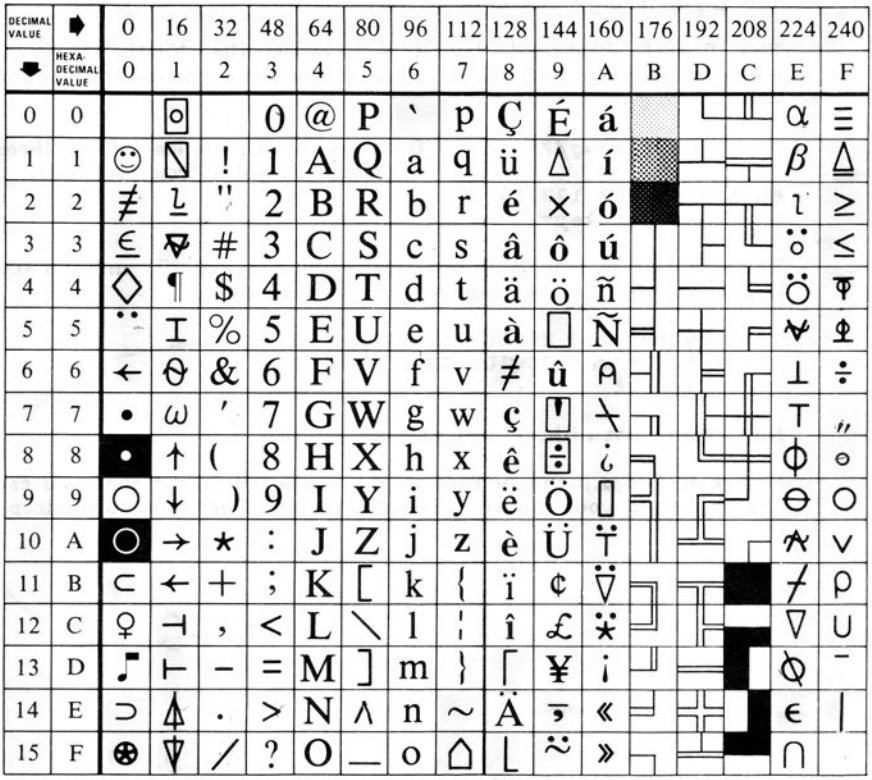


**Figure B-2: IBM character ROM**

**Characters that differ from the APL ROM are shaded.**

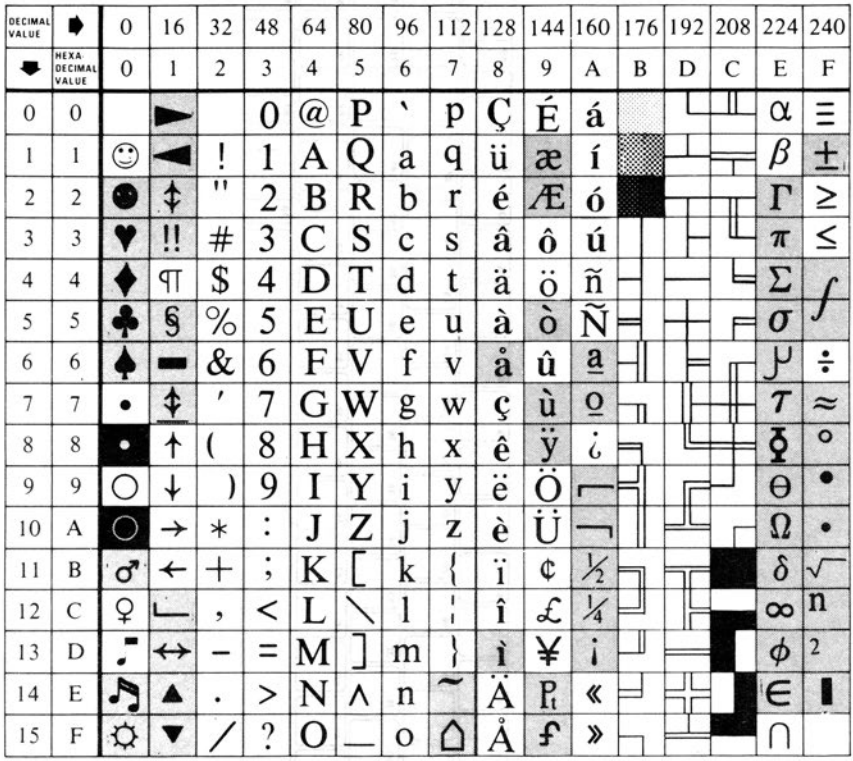


Figure B-3: Key numbers

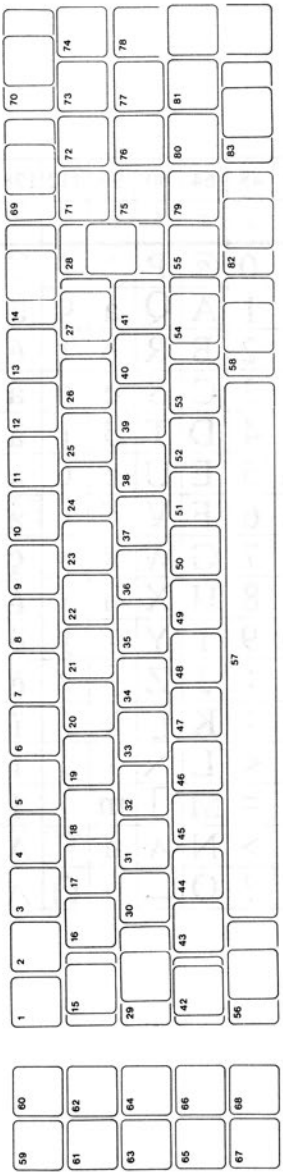
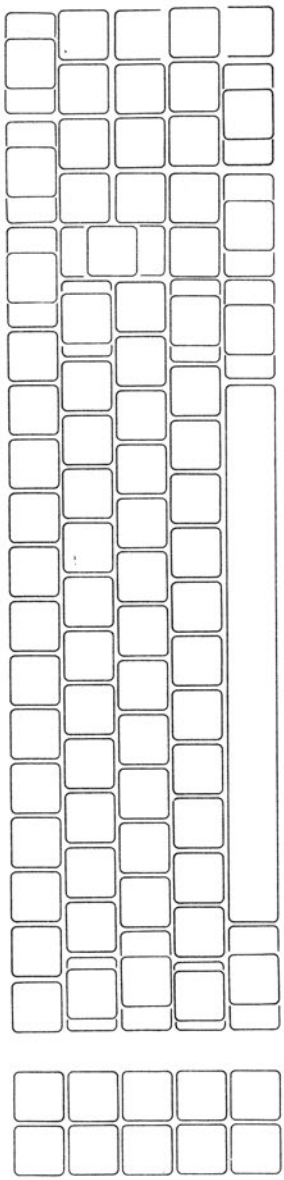


**Figure B-4: Blank keyboard**



## SESSION MANAGER ACTIONS

In the keystroke table, a negative value indicates that the keystroke performs a Session Manager action. In the subcommand *x* to *□arbout* 3 22 5 5 *,x*, the corresponding absolute values perform the same Session Manager action. These are all such values, and their actions:

**Immediate actions** (0-63)

2 <Alt down>
moves the current window down one row, that is, toward newer lines in the session log.

3 <Alt up>
moves the current window up one row, that is, toward older lines in the session log.

4 <Alt PgDn>
moves the status line down one row, thus opening the upper window by one more row.

5 <Alt PgUp>
moves the status line up one row, thus closing the upper window by one more row.

6 <Alt keypad-5>
selects the other window, that is, switches the window selection from lower (default) to upper, or upper to lower.

7 <ScrollLock>
locks/unlocks scrolling of the lower window. Default: unlocked.

8 <Alt right>
moves the current window right one column; ignored unless wrap disabled by action 26.

9 <Alt left>
moves the current window left one column; ignored unless wrap disabled by action 26.

10
allows a curbed program to emit one output line.

11 <shift NumLock>
enables/disables the keyboard alarm bell. Default: disabled.

12 <shift down>
moves the current window down one page.

13 <shift up>
moves the current window up one page.

14 <Ctrl Del>
resets the keyboard: clears the typeahead buffer; turns off caps reversal; resets insert/replace mode to insert; deactivates numeric keypad.

15 <Ctrl Esc>
enters DEBUG.

16 <shift right>
moves the current window right a quarter of its width.

17 <shift left>
moves the current window left a quarter of its width.

19
marks the peg as the upper bound of the deletable portion of the session log; that is, action 20 will act on all portions of the session log **newer** than where the peg is at action 19.

20
deletes the session log between the current peg and the deletion marker set by action 19.

21
resets the display: moves the peg off screen; sets wrap on both windows; returns the upper window to the oldest lines and the lower to the newest; resets current-window status to the lower window; unlocks lower-window scrolling.

22 <Alt Home>
moves the current window to show the oldest line in the session log.

23 <Alt End>
moves the current window to show the newest line in the session log. If the current window is the lower window, the weak scroll state ends; if the input window subsequently expands, the newest line remains visible.

24
forces a repaint of the screen. **Used only internally.**

25
changes the peg's viewing attribute, in the order: reverse (default), blinking, regular, intense.

26 <Alt NumLock>
enables/disables wrapping of lines in current window. When wrapping is enabled, actions 8, 9, 16, and 17 are ignored. Default: enabled.

27 <Esc>
resets the Session Manager completely; it combines the effects of Session Manager actions 14 and 21.

28
scrolls the current window one **line** down. Compare this with action 2, which scrolls the window by **screen row.**

29
scrolls the current window one **line** up. Compare this with action 3, which scrolls the window by **screen row.**

30 <shift Home>
moves the current window to show the reference line.

31
moves the current window to show the second reference line.

32 <shift keypad-5>
sets the reference line at the current peg.

33
sets the second reference line at the current peg.

34
scrolls the current window up half its depth.

35
scrolls the current window down half its depth.

36 <PgUp>
moves the peg up one line.

37 <PgDn>
moves the peg down one line.

38
curbs the executing APL program, causing it to pause at the next output attempt. Overridden by action 39 or 40.

39
causes an APL program to resume from a curbed condition. Overrides Session Manager action 38 or 40.

40 <Alt ScrollLock>
curbs or uncurbs the executing APL program; the curb toggle. Overrides Session Manager action 38 or 39 or itself.

**Editing actions** (64-127)

64 <return>, <Alt return>, <shift return>
enters the current input line (sends it to APL); that is, ends input mode. Compare this with action 81.

65 <shift ScrollLock>
signals *input interrupt* (similar to SHARP APL/370's *o* BACKSPACE *u* BACKSPACE *t* NEWLINE).

66 <backspace>, <shift backspace>
moves the cursor left one column, destroying the character that was there.

67 <Del>
deletes the character at the cursor and shifts the remainder of the input line leftward.

68 <left>
moves the cursor left one column.

69 <right>
moves the cursor right one column.

70 <Home>
moves the cursor to the first column of the input window.

71 <End>
moves the cursor to the end of the input line.

72 <Alt backspace>
forms a single overstrike by backspacing the cursor and allowing you to type one more graphic. If you try to form an illegal overstrike, the Session Manager waits for a legal overstrike. (You can legally overstrike any character with <space>.)

73
enters multiple-overstrike mode. Illegal overstrikes are treated as by action 72.

74
enters multiple-overstrike mode. If you enter an illegal overstrike, the later character remains.

75 <up>
moves the cursor up one row.

76 <down>
moves the cursor down one row.

77 <Alt Del>
deletes everything from the cursor through the end of the input line.

78 <keypad-minus>
copies the peg to the input stage, completely replacing what's there.

79 <shift tab>
moves the cursor left eight columns.

80 <tab>
moves the cursor right eight columns.

82 <shift Del>
deletes all characters to the left of the cursor.

83 <Ins>
toggles insert/replace mode.

84 <keypad-plus>
copies the peg to the cursor, inserting it or replacing the input text, according to whether the keyboard is in insert or replace mode.

85
copies the reference line into the input window, replacing the entire input line.

86
copies the reference line to the cursor, inserting it or replacing the input text, according to whether the keyboard is in insert or replace mode.

87
deletes entire input stage.

## SUMMARY OF □*arbin* AND □*arbout* CODES

□*arbin* 3 22 5,cmd,scmd,data

3 22 5
is the header for arbitrary output to the Session Manager.

**cmd**
indicates the Session Manager command. **cmd** may be one of: 2, 4, 5, 6, 8, 9, 10.

Command 2 indicates: read or set hardware character table.

Command 4 indicates: read or set keystroke table.

Command 5 indicates: do a Session Manager action.

Command 6 indicates: read or set function key table.

Command 8 indicates: send the following data to LPT1.

Command 9 indicates: read or set overstrike table.

Command 10 indicates: next entry to be invisible.

**scmd**
indicates the subcommand. For commands 2, 4, 6, and 9, the subcommand is either 0 for read, or 1 for set. For command 5, the subcommand is a Session Manager action; see **Session manager actions** above. For command 8, the subcommand is 2.

**data**
is data for the command. Data has meaning only for subcommand 1 of commands 2, 4, 6, and 9, and for subcommand 2 of command 8.

For subcommand 2 of command 8, **data** is sent as is to LPT1; for most printers, it is ASCII character codes, and unique escape characters. For best results, end **data** with 10 13, that is, CARRIAGE RETURN, LINEFEED. In any case, consult the printer's manual.

For more information about the hardware character, keystroke, and overstrike tables, see the sections earlier in this appendix. For more information about the function key table or about invisible output, see chapter 6 **Session Manager**.

## Appendix C: UNION KEYBOARD AND UNION CHARACTER SET

With SHARP APL for the IBM Personal Computer, I.P. Sharp Associates introduced a new arrangement of the APL keyboard: the **union keyboard**. It combines in a single layout all the characters of an ordinary typewriter or ASCII terminal and all the additional characters used in APL programs.

For people who know how to type but haven't used APL previously, the new keyboard is a great deal easier to use. It also enables natural-language text and APL to be intermixed without having to switch out of one character set into another.

At present, the new keyboard is only available on devices which have soft, that is, programmable keyboards and displays, such as the IBM PC. However, union keyboards will likely become available for a variety of terminals, as well as for standalone APL systems.

The change to the keyboard--merging APL characters into a single expanded set--is complemented by a parallel change to the set of characters recognized by the APL system. In APL, the set of all possible characters is called *⎕av*. It now contains a distinct position dedicated to each of the standard ASCII characters and APL symbols.

**Why the change? And why now?** Two separate keyboards--one for APL and a different one for everything else--has long been a nuisance, especially for people who must regularly switch back and forth between them. Moreover, the APL keyboard lacked many characters needed to process ordinary text. For years this has complicated programs that dealt with text, and made it harder to set up displays intended for ordinary terminals.

Consequently, I.P. Sharp has changed the keyboard now that the technology is available to combine APL and ASCII displays easily. APL developers would have preferred to have done this all along: to integrate APL with standard characters and keyboards. But until recently, it wasn't possible to do this.

So the real issue is not Why the change? but Why not sooner?

**Early hardware limitations.** APL was first used on a timesharing system in 1965. Adin Falkoff and Ken Iverson at IBM Research designed a special APL typeball for the terminals then available. Its mechanism was based on the IBM Selectric typewriter. The ball could accommodate 88 characters (the base and the shift positions of each of the 44 keys). But to merge APL with the existing typewriter characters would have taken well over 150 characters. How was it possible to squeeze them all in? They managed to get everything they needed for APL onto the ball by the following economies:

1. They constructed many of the APL characters by overstriking pairs of characters already there. This permitted ⊛ to be formed from ○ and ⋆. (Overstriking had another long-term benefit: it was possible to add new characters later without building a new typeball.)

2. They left out all the special characters found on a conventional typewriter but not used in APL programming. That freed the spaces ordinarily used for such characters as @ # $ % &. (Leaving them off the typeball meant leaving them out of ⎕*av* as well.) Although you could manually replace one typeball with another, the terminal could not tell the computer which one was in use. It transmitted a code to the computer describing only which **key** you pressed, so there was no way of saying "This key means APL =", or "This key now means ASCII %".
3. They saved 26 positions by electing to have only one position for each letter of the alphabet, instead of two. Ordinarily, a typeball would have both a minuscule (lower case) and a majuscule (upper case) form for each letter. They decided instead to have just one form, and to obtain a second by overstriking.

**The two forms of the alphabet.** Since the Middle Ages, European languages have used two forms of each letter: minuscule, in the body of a document, and majuscule, for emphasis. Because the emphasis alphabet is used less frequently, typewriters (and ASCII terminals) have minuscule in the base position, and majuscule in the shift position, which requires pressing an additional key. There is a clear tradition that minuscule is the work-horse and majuscule is for special effect. (High-school algebra uses minuscule letters for the names of variables, and so did Iverson's book **A Programming Language**.)

However, there is also the tradition that when you can write a letter only one way, you use the majuscule (which was the form the Romans gave us, before each letter had two forms). So you see majuscule-only in telegrams. Falkoff and Iverson's typeball adopted as its primary alphabet (keyed in the base position) the majuscule letters rather than the minuscules that those keys ordinarily produce. For emphasis, you could underline them.

**Smart terminals and three-case keyboards.** A microcomputer (which may be used as a standalone terminal, or as a terminal to a host system) usually has what amounts to a third case, labelled **Alt**. (ASCII terminals also have a special shift called **Ctrl**; but, as its name implies, it is used for control signals. Moreover, it wasn't available for all keys because there weren't enough distinct codes in the seven-bit transmission normally used for ASCII.)

But on a micro and other ASCII terminals, <Alt> case can be available for every key (and so can <Ctrl>, if need be) and the number of keys is 47 instead of 44. So there are at least 141 distinct characters possible, not to mention those that can still be formed by overstriking. APL no longer need be confined within the 88 positions of the Selectric typeball! The union keyboard and union character set are therefore based on the following reforms:

1. Restore the missing ASCII characters @ # $ % & etc. to ⎕*av*. This has already been done in both the System/370 and Personal Computer versions of SHARP APL.
2. Keep the existing ASCII significance of every key on the keyboard. This means that the APL characters

() [] +-= :; ' are found in the same positions as on the standard ASCII keyboard. For example, <shift slash> (at the bottom right corner of the keyboard) is a question mark in ASCII, and so it is ? in APL also.

3. Use the Alt case to obtain those APL characters not available in ASCII. As far as possible, locate them in the same position as in the old APL-only keyboard. Thus ⍴ is formed by <Alt R>, ⍵ by <Alt W>, and so on. Permit overstriking to form APL characters in the same way as before. Thus ⌸ can still be formed by overstriking ⌷ and ⌸.
4. Restore the original use of minuscule and majuscule letters. That is, the minuscules *abcdefg* are in the primary alphabet (⎕*av* elements 86 to 111), while majuscules *ABCDEF* are in the secondary (or emphasis) alphabet (⎕*av* elements 113 to 138).

**Psychological conversion.** The new arrangement is a great deal easier for newcomers to APL, since they no longer have to unlearn any typing habits. They only have to learn the locations of the additional characters used in APL.

Users already experienced with the old APL keyboards may find the shift awkward for a while, but not for very long. The only thing they must learn is to revert to the familiar standard keyboard.

The greatest difficulty is experienced by those users who must continue to go back and forth repeatedly between the old and new keyboards. But they are already accustomed to doing this, if they have ever used a typewriter or an ordinary ASCII terminal.

**Data conversion.** Some users own files containing text data in which someone has carefully supplied the "wrong" APL characters so that they will look reasonable when displayed on an ASCII-only device. For example, the stored data might contain the APL characters:

```
∘ohn ∨'ing 1199+1216∧[
son of Δenry ⍳⍳ > brother of ⍴ichard ⍳(
```

When transmitted to an ASCII-only device, those APL characters would appear as:

```
John (King 1199-1216);
son of Henry II & brother of Richard I:
```

Writing one set of APL characters in order to elicit different ASCII characters becomes unnecessary once we adopt a union keyboard and union character set. But a one-time conversion of the data is required. Functions to do such conversions will be available in the public libraries of SHARP APL/370 systems for those who wish to take advantage of them.

## Appendix D: DIFFERENCES BETWEEN SHARP APL'S

The SHARP APL interpreter that you use on your Personal Computer is identical to the SHARP APL interpreter at the heart of a large timeshared APL system serving hundreds of users from a System/370 mainframe computer. The internal representation of numbers and characters is also identical. When you perform a calculation in SHARP APL/370 and then repeat it in SHARP APL/PC, the results you get are exactly the same. In SHARP APL/PC, the only restrictions on the rank of data or the length of data axes are those that exist also in SHARP APL/370: the size of an array is limited only by the total workspace available, and the number of axes can't exceed 63.

Nevertheless, the environment in which SHARP APL exists on the IBM PC is very different from the environment of work from a terminal connected to SHARP APL/370. This appendix discusses the reasons for those differences. Points of difference are also noted in the **SHARP APL Pocket Reference**. Depending on how an application is written, the practical effects that these differences have on your work or your use of a particular application program may be trivial or extensive.

From the viewpoint of a user of SHARP APL/PC, the differences can be summarized by the following observations:

- You are the Operator.
- At any particular moment, you are the machine's only user.
- Only one APL task can run at a time.
- The size of the workspace on each PC is what's left over after you subtract the APL system from the total memory.
- There are no session variables.

Each of these observations is expanded in turn.

### You are the Operator

A data centre running SHARP APL/370 is operated by a team of professionals. You may never see them, but they're there, starting APL, making adjustments, restarting the system when it fails, backing up files or workspaces in case you may need them later (and retrieving them for you when you do), maintaining data bases, running auxiliary processors that you may need, managing the equipment on which your data files are kept, and even answering your questions. On the PC, **you** are the operator. This has some important consequences:

- There's no provision to save your current work and restore it after a system failure (other than what you can do yourself to save workspaces or write data to files). And it's your job to keep the diskettes containing your data in a safe place. SHARP APL/370 sets aside a copy of your current work every few seconds so that a system failure causes little loss. That isn't feasible in a desktop computer.

- You're in charge of the system clock. Workspaces and file components are timestamped in SHARP APL/PC, as they are in

SHARP APL/370. But on a timesharing service, the operations staff sees to it that the system clock is set properly. On a PC, you have to tell the PC operating system the date and time each time you start the machine. You may have an option that keeps the clock going while power is off, and tells the computer what time it is whenever the computer starts. But in either case, the time and date are no more accurate that the input you provide.

- If you want files or workspaces backed up, you have to do it yourself.

- If you want an APL file rewritten, you have to do it yourself, using an APL program. (It is useful to do this if some components have been replaced by values that take more space than the values previously stored, or if components have been dropped from the file; the holes left by these changes don't become available for other use until the file is copied and the old one erased.) On a SHARP APL/370 system, many users aren't even aware that rewriting the files to optimize storage is something the operations staff does routinely.

- There's no system command to send a message to the Operator.

**At any particular moment, you are the machine's only user**

"Library" in the context of the PC is the box of diskettes you bring with you to the machine, and the particular disks or diskettes you mount at the time you are using it.

For users of SHARP APL/370, the work of many people--sometimes thousands--is in a common storage. SHARP APL/370 provides means to sort out and regulate which user has access to which workspaces or files. Storage is partitioned by **library number** (usually the user number of the owner of each file or workspace). Users can arrange to make use of files or workspaces belonging to others, or created by the system managers for public use.

In SHARP APL/PC, storage of APL files or workspaces is not grouped or classified by the user number of its owner. Instead, whoever has access to a diskette or hard disk has access to whatever's on it. In effect, "library" on the PC is what you can get your hands on.

- The absence on the PC of individual libraries securely linked to password-protected user numbers can be partially replaced with the use of different directories for different users.

- In SHARP APL/PC, you refer to disk drives in much the same way as timesharing users refer to libraries. On the PC, the command *)lib* 1 asks for a list of the SHARP APL workspaces on the disk now mounted on drive 1 (known to DOS as drive A); *)lib* 2 asks for those on drive 2, and so on. Similarly, *□lib* 1 returns a list of the SHARP APL files on the disk now mounted on drive 1. Drive 0 means the default drive. When you don't indicate a drive explicitly, the current drive is assumed.

- Filehold aren't needed. In SHARP APL/370, it's possible for several users to use the same file at the same time. Because two of them might try to update it at once, an application can use the function □*hold* or □*fhold* to verify that a file is free and then to hold it for exclusive use during a critical segment of a program. In SHARP APL/PC, there's no risk of conflict between concurrent users since no more than one can be signed on at any time. On the PC, the filehold functions are simply no-ops: the File System doesn't have to do anything to enforce them.

- The function □*rename* can change a file's name, but not the disk on which it's located. In SHARP APL/370, a different library number can be part of the new name. In SHARP APL/PC, the same phrase means a different drive rather than a different library, which implies physically moving the file, not just renaming it. That isn't possible.

**Only one APL task can run at a time**

There are no S-tasks or N-tasks or B-tasks.

In SHARP APL/370, you start an S-task by offering to share a variable with processor 1. But the Personal Computer has no processor 1, so the offer is never accepted.

In SHARP APL/370, you start an N-task by executing □*run* with an argument indicating how the task is to be run. In SHARP APL/PC, the result is always 2 0 (capacity for N-tasks exhausted).

In SHARP APL/370, you start a B-task by loading a workspace which then transmits your request by sharing a variable with the B-task scheduler. Neither that workspace nor the B-task scheduler is supplied with SHARP APL/PC.

The absence of facilities for N-tasks, B-tasks, or S-tasks probably means that a program imported from the mainframe which depends on them will require some adaptation before it will run on a PC.

**Workspace size is what's left after you subtract APL from your PC's memory**

On most Personal Computers, the workspace size available to you in SHARP APL/PC may be greater or smaller than in SHARP APL/370. It may be different from the next PC you use, and different from your own next week or last week (if you add or subtract memory boards or change APs or start paramaters). This means that there is no guarantee that a workspace you saved on one machine can be loaded onto the one you're using now, or that the programs will have enough room to run if you load it.

However, the only restriction on loading a workspace is that it must fit in your current workspace size.

**There are no session variables**

You can set a value for the variable □*sp* in an active workspace, and include it in a saved workspace. But □*sp* from the active workspace does not survive anything that clears the active workspace (*)load*, *)xload*, □*load*, □*qload*, or *)clear*).

The variable □*ht* is used in SHARP APL/370 to help the APL system to understand how input from a terminal may be spaced when the terminal has tab stops set at fixed positions. The IBM PC does not have such tab stops. So you can set □*ht* but the SHARP APL/PC system ignores it, and does not propagate it into a new workspace.

## CHECKLIST OF DIFFERENCES

| Topic | Behaviour on the PC |
|---|---|
| User number | Start parameter; see chapter 2. |
| Characters | The first alphabet is displayed as lowercase and the second alphabet as capitals. The third-alphabet positions in ⎕*av* are displayed in the same way as the first; and they are not (by default) enterable from the keyboard. |
| Names | Names of files and workspaces are restricted to 8 characters (rather than 11) and may not include characters from the second alphabet. If qualified by a preceding number, the number is interpreted as drive number rather than library number. |
| Library | Organized by disk drive and current directory rather than by user number. For example, the argument of ⎕*lib* refers to drive number (with 0 meaning the current drive). |
| ⎕*run* | Result is always 2 0 (system capacity for N-tasks exhausted). |
| ⎕*runs* | Result always has a single row. Element 1 (task identifier) is always 1; element 3 (initiator) is always 8. |
| 1 ⎕*svo* shv | Share never matched because there is no S-task processor. |
| 2 ⎕*ws* 3 | Elements 10 and 11 (port and task identifier) are always 1. Element 12 (device) is 8 (PC). |
| ⎕*ai* | Elements 6, 7, and 9 contain the same values as elements 2, 3, and 5, respectively. (On SHARP APL/370, 6, 7, and 9, are the values for 2, 3, and 5 cumulatively since the last system billing.) |
| ⎕*ht* | Not used, and not propagated when workspace loaded or cleared. |
| ⎕*pw* | Default is 80. |
| ⎕*sp* | Not propagated when workspace loaded or cleared. |
| ⎕*av* | Elements 241-251 (origin 0): single-line box-drawing characters (available in SHARP APL/370 only for IBM 3270 display stations). |
| ⎕*arbin*/⎕*arbout* | Used to communicate device-specific control information between the APL workspace and the device managing the APL session. The meaning and |

| | |
|---|---|
| | effect of the codes differ widely among the many devices usable in SHARP APL/370, and the uses on the PC are different from any of these. |
| □*drop* | Does not free space in the PC's mass storage. (File can be squeezed by copying.) |
| )*copy* | *ws full* likelier than with workspaces of equivalent size in SHARP APL/370. The entire source workspace must fit in available mainstore along with the entire sink workspace. |
| )*cpulim* | Not supported. |
| )*keyboard* | Not supported. |
| )*lib* | Returns workspace names unsorted, according to directory order. "Library" is actually current directory of specified or implied disk drive. |
| )*msg* | Not supported. |
| )*opr* | Not supported. |
| )*ports* | Not supported. |
| )*terminal* | Not supported. |

## Appendix E: BIBLIOGRAPHY

This appendix lists some books you might find useful. It includes a variety of sources on the APL language or its applications.

Titles marked **I.P. Sharp** are available from the Publications Department, I.P. Sharp Associates Limited, 2 First Canadian Place, Suite 1900, Toronto, Ontario, Canada M5X 1E3, or from any I.P. Sharp office.

Titles marked **APL Press** are available from APL Press, 220 California Avenue, Suite 201, Palo Alto, California, USA 94306-1683.

### Expressly for the IBM PC

**Introduction to APL.** K.E. Iverson, 1984. Do-it-yourself text, together with diskette containing workspaces executable in SHARP APL/PC, APL*PLUS/PC, and IBM's APL for the PC. APL Press.

### General texts on APL

**APL Language.** IBM publication GC26-3847. This combined introduction and reference is intended for IBM's APL, but the description of the language applies just as well to SHARP APL or to almost any other implementation of APL.

**APL: An Interactive Approach.** Leonard Gilman and Allen J. Rose. John Wiley & Sons, Inc., New York. This text has been through several editions; the first was specific to APL*PLUS, but later ones make at least passing reference to other dialects. The third edition, 1984, includes occasional mention of STSC's APL for the PC, and some of the extensions peculiar to SHARP APL, but omits mention of such major features as complex arithmetic, the composition operators, packages, or event-handling. A Spanish version, **APL: Enfoque Interactivo** was published in 1982 by LIMUSA publishers, Balderas 95, Mexico 1 DF, Mexico. A German version was published as **APL anwendug IBM-systemen**, Frankfurt, Wiley GmbH, 1973. A Russian version was published in the USSR as **KYRC APL** in 1979.

**APL: The Language and Its Usage.** Raymond P. Polivka and Sandra Pakin, 1975, Prentice-Hall, Englwood Cliffs, New Jersey, USA. A full-sized text, based on timesharing APL, but most of it equally relevant to APL on the PC.

**APL: A Design Handbook for Commercial Systems.** Adrian Smith, 1982. Practical discussion of a great many issues in the design of working APL programs in a business setting. John Wiley & Sons, Chichester, England.

**In French:**

**APL: une approche pratique**. Maurice Dalois, 2nd ed., 1981. Softcover, complete text, with exercises and examples. A few examples specific to APLSV or APL*PLUS, but generally applicable to any APL system. Ministere de l'Education, Direction des Services informatiques, Gouvernement du Quebec, Canada.

**Apprendre et appliquer le langage APL**. B. Legrand, 1979. Paperback reproduction of typescript of a complete text, including both exercises and a separate section of answers to problems. Implementation-specific material mentions APLSV, VS APL under CMS, and APL*PLUS. Masson SA, 120 Boulevard St-Germain, 75280 Paris, France.

**In German:**

**APL II**. Wolfgang H. Janko, 1981. The "II" in the title refers to the level of the course (and not the dialect APL2). Two pocket-size paperback volumes, reproduction of typescript of college-level course. Athenaeum, Koenigstein, Germany.

**In Japanese:**

**APLSV Computer Language**, Junichi Osada and Akira Uchiyama, 1975. Although the title mentions APLSV, this hardcover text is a general introduction to APL. Maruzen, 2-3-10 Nihon-bashi, Chuo-ku, Tokyo.

**In Dutch:**

**Programmeren in APL**. H.F.F. Uyttenhove, 1983. A concise course in APL, with exercises. It overviews the most common primitives, discusses how to build functions, and how objects are organized in a workspace. Samson Uitgeverij BV, Stadhoudersplein 1, Postbus 4, 2400 MA Alphen aan den Rijn, The Netherlands.

**Historical background**

**A Source-Book in APL**, A.D. Falkoff and K.E. Iverson, 1981. Eight of their papers describing the development of APL, originally published in various journals between 1964 and 1980. APL Press.

**APL Quote-Quad: The Early Years**. The first 14 issues, spanning 1969-1972 (otherwise long out of print) reproduced in one volume. APL Press.

**General topic treated in APL**

**Elementary Analysis**. K. E. Iverson, 1976. A continuation of the approach developed in **Algebra** to cover polynomials, slopes, and the exponential and triogonometric functions, using APL notation which can be directly executed. With exercises. Paperback; APL Press.

**Calculus in a New Key**. D. L. Orth, 1976. Treatment of derivatives and integrals as tools for analysing and constructing functions, with APL as the exclusive notation throughout. Paperback; APL Press. Also available in a Japanese hardcover translation (as **APL-algorithmic approach to Calculus**) from Kogaku Tosho (publishers), 5 Sanban-cho, Chiyoda-ku, Tokyo.

**Resistive Circuit Theory**. Robert Spence, 1979. College-level introduction to circuit theory, with APL as the notation for both presentation and computation. APL Press.

**Accounting Structured in APL**. Yuji Ijiri, 1984. A monograph in the form of question-and-answer dialogue between students studying accounting, with examples. Accounting Education Series, vol. 6, American Accounting Association, 5717 Bessie Drive, Sarasota, Florida, USA 33583.

**Algebra: An Algorithmic Treatment**. K.E. Iverson, 1972. Topics for the first or second-year course in secondary-school algebra, with emphasis on APL as the notation, and experimentation with arrays as a major activity. With exercises; answers are available as a separate publication. Paperback; APL Press.

**APL-Stat: A Do-it-Yourself Guide to Computational Statistics Using APL**. James B. Ramsey and Gerald L. Musgrave, 1981. Full-size softcover text, including both instruction on statistics and introduction to APL. Examples are based both on timesharing system and use of the IBM 5120 minicomputer version of APLSV. Lifetime Learning, a divison of Wadsworth, Inc., Belmont, California, USA 94002.

**Computing in Statistical Science through APL**. Francis John Anscombe, 1981. An introduction to APL with many brief statistical examples, followed by three long chapters giving a detailed mathematical, statistical, and computational discussion of a substantial practical problem. Anscombe aims to show how the statistician can control a computer so that it becomes an extension of himself, as the piano is to the pianist. Springer-Verlag, New York.

**Mathematical Experiments on the Computer**. Ulf Grenander, 1982. From the Preface: "The main thrust of the book is how to carry out mathematical experiments . . . quickly and easily, using the computer only as a tool, and with the emphasis on mathematics rather than on programming. For the purpose, interactive computing is almost indispensable, and . . . . the programming language APL has no existing rival that can seriously compete with it for use in mathematical experiments." The book includes "A Mathematician's Guide to APL" and the definitions of more than a hundred APL functions. Academic Press, Inc., 111 Fifth Avenue, New York, USA 10003.

### Conference proceedings

A conference on APL has been held more or less annually since 1969. Since 1979, the principal series has been under the sponsorship of the Association for Computing Machinery, and ACM has published the proceedings. Publications may be ordered from ACM at P.O. Box 64145, Baltimore, Maryland, USA 21264. The association's headquarters are at 11 West 42nd Street, New York, USA 10036. Earlier proceedings were published on an ad hoc basis by the committees that organized them.

In addition, I.P. Sharp Associates and STSC Inc. have each sponsored APL users meetings whose proceedings have been published.

**APL Users Meeting, 1978.** I.P. Sharp, Toronto.

**APL Users Meeting, 1980.** I.P. Sharp, Toronto.

**APL Users Meeting, 1982.** Vol 1: Applications. Vol 2: Technical Topics and Applications Design. I.P. Sharp, Toronto.

**The Information Centre and Changing Technologies: Proceedings of the 1984 APL Users Meeting.** I.P. Sharp, Toronto.

**APL in Practice.** Proceedings of the STSC APL Users Meeting, 1980. John Wiley & Sons, New York.

### Periodicals

**APL Quote-Quad** is published more or less quarterly by the Special Interest Group on APL of the Association for Computing Machinery (SIGAPL) and comes automatically with membership. Nonmembers may also subscribe; you can get information from the coordinator of SIG/SICs at ACM headquarters in New York.

**I.P. Sharp Newsletter** appears every two months. It includes both news regarding SHARP APL, and technical supplements discussing programming issues and techniques. Subscription can be arranged through any I.P. Sharp office or by returning your SHARP APL/PC registration card.

### Lightweight

**APL in Exposition.** K.E. Iverson, 1973, APL Press.

**Introducing APL to Teachers.** K.E. Iverson, 1973, APL Press.

**Introducing APL to Scientists and Engineers.** K.E. Iverson, 1973, APL Press.

**Understanding APL.** Susan Bryson, 1982, Alfred Publishing Co., PO Box 5964, Sherman Oaks, California, USA 91413.

**Starmap.** P.C. Berry and John Thorstensen, 1973. Modular programming in APL to calculate the coordinates of the stars and planets as seen at any particular time, date, and location on the Earth. Includes all needed functions and data, but not the graphics software. APL Press.

**The Four-Cube Problem**, E.E. McDonnell, 1981. Solution to "instant insanity" in an applicative style of APL, and comparison with a prize-winning BASIC program that does the same thing. APL Press.

**Finn-APL Idiom Library**, 1982. Collected handy APL expressions. APL Press.

**APL and Insight.** Paul Berry, Giovanni Bartoli, Carlo Dell'Aquila, and Vincenzo Spadavecchia, 1970. Discusses the use of programming to convey mathematical concepts to pupils.

### About the PC

**Disk Operating System User's Guide by Microsoft Corporation.** IBM publication 1502344. A step-by-step guide for the new user of DOS. It describes how to start using DOS, how to use the most useful DOS commands, and what likely DOS messages mean.

# Appendix F: DISTRIBUTED WORKSPACE *utility*

This appendix lists and describes the utility software distributed as workspace *utility* with SHARP APL/PC. This appendix is intended for programmers of end-user products; it assumes familiarity and experience with APL, and SHARP APL in particular.

You can find each of these functions in distributed workspace *utility*. Most of the functions in *utility* have been extracted from the SHARP APL Utility Library, 2 *ulibrary*.

**The function titles** are given in this format:

qualifier/name.v

**qualifier**
indicates where the function came from (usually the author's I.P. Sharp identification). This qualifier is an important aid for retrieval from the Utility Library. Anything to the left of the last slash is qualifier.

**name**
is the function's actual name in the *utility* workspace.

**v**
is the function's version number in the Utility Library.

Workspace *utility* is quite large--about 80K--and may give *ws full* when it is loaded or copied. If this happens, bring APL up without any other processors, by executing CP8X only. This should allow you to load or copy the workspace, and save smaller workspaces containing only the objects you need.

**Title:** *cdb/calendar.2*

**Summary:** Returns calendar for year as 12 element enclosed vector.

**Description:**
Returns calendar for given year as a 12 element vector; each element is an enclosed character matrix, appropriately formatted. The right argument is the year (only the last 2 digits are needed if in this century). For example, the calendar for January and February 1984 is:

```
      (calendar 84)[⍳2]
         jan                     feb
 su mo tu we th fr sa   su mo tu we th fr sa
  1  2  3  4  5  6  7                  1  2  3  4
  8  9 10 11 12 13 14    5  6  7  8  9 10 11
 15 16 17 18 19 20 21   12 13 14 15 16 17 18
 22 23 24 25 26 27 28   19 20 21 22 23 24 25
 29 30 31               26 27 28 29
```

---

**Title:** *cdb/fncompare.1*

**Summary:** Compare two functions

**Description:**
Syntax: *r←fn1 fncompare fn2*

Compares the ⎕*cr* representations of functions, returning "*no difference*" or (vector) text of mismatches and swapped lines. Arguments may be either (nonmatrix) function names, or ⎕*cr*s of functions. For example:

```
      ∇ foo                          ∇ goo
[1]   ⍝ test <fncompare>     [1]     lineb
[2]     linea                [2]     linec
[3]     lineb                [3]     linea
[4]     linec                      ∇
      ∇
```

```
      'foo'fncompare 'goo'
1 [0] foo
2 [0] goo
1 [1] ⍝ test <fncompare>
1 [2] linea
2 [3] linea
```

Showing: 1. different function names; 2. line [1] of *foo* not found in *goo*; 3. the line *linea* found in both functions occurs before the other two lines in *foo*, but after them in *goo*.

---

**Title:** *djk/cpu.1*

**Summary:** Measures the number of CPU units required to execute expressions.

**Description:**
*milliunits←repetitions cpu expressions*

*cpu* measures how many milliunits are required to execute one or more expressions, as in the following example.

```
      10 cpu '⍳1000'⊃'+/⍳1000'⊃'+/0.1+⍳1000'
22.4 35 84.7
```

The right argument should be a vector of enclosed character vectors. The left argument is the number of times each expression is to be executed. The result is a numeric vector indicating how many milliunits are required to execute each expression **once**. The result is inherently approximate; increasing the number of repetitions will usually increase the accuracy of the result. Note that the result can vary widely depending on the workspace in which *cpu* is executed.

Overhead incurred by *cpu* is not isolated out of the result. Thus the actual number of milliunits used can be obtained via *(n cpu expressions)-n cpu* '0'.

Each expression may be any APL expression, but not a statement, i.e. it must return a result and may not contain diamonds.

---

**Title:** *djk/datatype.1*

**Summary:** Determines the internal data type of an array (or package).

**Description:**
*type←datatype x*

Determines the internal datatype of *x*. *x* may be anything. The result is one of the following: '*bool*', '*char*', '*int* ', '*flpt*', '*cplx*', '*encl*', and '*pack*'. E.g. *datatype* 0⍴1.2 ↔ '*flpt*'.

---

---

**Title:** *djk/each.2*

**Summary:** E.g. *'foo ⍵*2' each 5⊃6⊃<⍳7 ↔ (foo 5*2)⊃(foo 6*2)⊃<foo (⍳7)*2.*

**Description:**
*result←α"> ⍵* where *⍵* is an array and *α* is a monadic direct-definition expression (**not** a statement--i.e. it may not contain diamonds).

Examples:

```
      ⎕io←1
      ⎕ps←¯1 ¯1 0 ¯3

      ⍴⎕←'⍳⍵' each ⍳4
|1| |1 2| |1 2 3| |1 2 3 4|
4
      ⍴'yourfn ⍵' each array ↔ ⍴array
      'yourfn ⍵' each ,array
  ↔ (yourfn >array[1])⊃(yourfn >array[2])⊃ ... ⊃<yourfn >a
rray[n]
```

---

**Title:** *djk/endspose.1*

**Summary:** Moves specified axes to the end of the shape vector.

**Description:**
*transposed←axes endspose array*

Moves the specified axes to the end of the shape vector.

Examples:

```
      ⎕io←1
      ⍴1 endspose 1 2 3 4 ⍴'a'
2 3 4 1
      ⍴3 2 endspose 9 8 7 6 5 ⍴⍳1001
9 6 5 7 8
      ⎕io←0
      ⍴0 endspose ⍳8
8
```

---

---

**Title:** *djk/fdelete.1*

**Summary:** Deletes specified components from a file.

**Description:**
*tienum fdelete compnums*

Deletes component numbers *compnums* from file number *tienum*; finishes with *⎕drop tienum,-⍴,compnums*. For example, if file 7 had 21 components and you wanted to delete componets 3, 9, and 17, you would execute the following:

```
      7 fdelete 3 9 17
```

After executing this, the file would have 18 components.

The elements of *compnums* should be distinct.

---

**Title:** *djk/randomize.1*

**Summary:** Finds a 'random' value for *⎕rl*.

**Description:**
*qrl←randomize*

Finds a 'random' value for *⎕rl*. The value, *qrl*, is dependent on *2↓⎕ts*. The range of values is essentially the domain of *⎕rl*, and each element of the range has about the same probability of being chosen.

---

**Title:** *djk/space.1*

**Summary:** Calculates an upper bound (within 1K) on the bytes required to execute an expression.

**Description:**
*bytes←space expression*

Calculates an upper bound on the number of bytes of workspace required to execute *expression*. The upper bound is within 1 Kbyte of the actual number of bytes required.

Examples:

```
      space '⍳1000'
4217
      space '0.1+⍳1000'
12392
```

*expression* may be any APL expression, but not a statement, i.e. it must return a result and may not contain diamonds.

---

**Title:** *glo/para.1*

**Summary:** Formats a vector of text to within a specified width.

**Description:**
*z←w para text*

Input specifications:
(1) *w* is a positive integer scalar.
(2) *text* is a character vector that does not begin with a blank, and contains no NEWLINEs, BACKSPACEs, or other nonenterable characters.

Output specifications:
(1) *z* is a character vector with embedded NEWLINEs, which appears as a paragraph when displayed.
(2) Each line of *z* contains as many complete words as fit on that line subject to the condition that if the last **nonblank** character (excluding the line-ending NEWLINE) is in position *p*, then $p \leq w$.
(3) If *text* contains a word of length greater than *w*, then that word will be broken so as to appear on consecutive lines of *z*.
(4) No line in the paragraph *z* commences with a blank.
(5) No characters are deleted from *text*. Thus, if the embedded NEWLINEs are removed from *z*, the result is identical with *text*.

Example:

```
text←'the quick brown fox jumped over the lazy dog while '
text←text,'the hue and cry went out for a '
text←text,'paragraph formatting program.'
```

*para* returns the results shown below. The jot ∘ marks the positions of the NEWLINEs.

```
      12 para text
the quick ∘
brown fox ∘
jumped over ∘
the lazy dog ∘
while the ∘
hue and cry ∘
went out for ∘
a paragraph ∘
formatting ∘
program.

      24 para text
the quick brown fox ∘
jumped over the lazy dog ∘
while the hue and cry ∘
went out for a paragraph ∘
formatting program.
```

⎕*io* is localized and set to 1.

*para* was written by George Lount; it won SHARP APL contest 3.

---

**Title:** *hui/sortlocal.1*

**Summary:** Sorts the local variables in a function header

**Description:**
Syntax: *z←sortlocal s*; *s* is the name of an unlocked function; *z* has the same results as □*fx*.

Example:

```
      ∇x←foo in;z;d;h;c;r;a
       ....
      ∇

      sortlocal 'foo'
foo
      ∇foo[□]∇
   ∇ x←foo in;a;c;d;h;r;z
[1]  ....
   ∇
```

*sortlocal* returns error messages for locked functions or a variable input.

---

**Title:** *lhg/Δrplc.1*

**Summary:** General replace of one string by another.

**Description:**
This function is a general string replacement utility. It replaces all occurrences of a particular target within a text vector by another string. Overlapping occurrences of the target string are ignored, so the resulting text vector always appears as expected.

The left argument to *Δrplc* is the text vector in which matches are to be sought and modified. The right argument is a two-element nested array containing the target string, followed by the replacement string. The result of the program is the final text vector, with all nonoverlapping occurrences of the target string replaced by the new string.

---

**Title:** *llf/listfns*.1

**Summary:** Formatted listing of functions named in right argument.

**Description:**
*listfns* provides a 1⎕*fd* listing of functions, with interfunction spacing and headers. The workspace library number and name are appended at the bottom of the list. It works in ⎕*io* 0 or 1; is protected against most erroneous right arguments the user may supply. The function names in the right argument may be supplied as a vector, with the names separated by blank or comma, or as a matrix with one function name per row.

---

**Title:** *mgf/pkgmatch*.1

**Summary:** As *r*←α≡*ω*, but also accepts packages as arguments.

**Description:**
*r*←α *pkgmatch ω* is identical with *r*←α≡*ω* for arrays. Additionally, it takes packages as arguments. Two packages are equal if they contain the same names and the same referents. This does not imply that 1 ↔ (⎕*pnames* α) ≡ ⎕*pnames ω*, since the order of the result of ⎕*pnames* is arbitrary.

If both packages contain one or more referents which are locked functions with the same name, and everything else is equal, the packages are considered equal. This is not the case if one package contains the same functions as the other, but locked.

*pkgmatch* is subject to ⎕*ct*.

**Subutilities:**
*mgf/pkgmatch*.1
*pcb/fnextract*.1

---

**Title:** *mgr/fnprint*.1

**Summary:** Prints the 1 ⎕*fd* of the supplied function name on the IBM PC printer.

**Description:**
Right argument: character vector containing a function name. Uses DOSX. Needs 500 bytes of SVP storage, but there is no limit on the function size.

---

**Title:** *mjab/bstranslate*.1

**Summary:** Changes character string with embedded backspaces used to represent overstrikes into true overstrikes.

**Description:**
Translates a vector of ASCII characters with embedded backspaces used to represent APL characters into a (shorter) vector of ⎕*av* elements and no embedded backspaces. Useful for translating something read from a screen or transmitted by some device into a form usable by 1 ⎕*fd*.

**Subutilities:**
*mjab/bstable*.2

**Title:** *mrab/format∆ts.1*

**Summary:** Converts *□ts*-style times into ISO-compatible character representations.

**Description:**
Syntax: *r←format∆ts ⍵*

Where *⍵* is an *n*-dimensional array of numbers containing *(×/¯1↓⍴⍵)* dates and times along its last dimension; each timestamp is a vector of integers in the usual *□ts* order, or five integers followed by a real number of seconds (with fractional part possibly nonzero, and *r* is a *(1⌈⍴⍴⍵)*-dimensional array of characters representing the corresponding dates in *⍵*.

Examples:

```
      ⍴□←format∆ts 1984 2 10 21 48 4 37
1984-02-10 21:48:04.037
23

      ⍴□←format∆ts 1984 2 10 21 48 4.037
                              Alternate form of first example
1984-02-10 21:48:04.037
23

      ⍴□←format∆ts 2 3⍴1984 2 2 1967 7 1
1984-02-02
1967-07-01
2 10

      ⍴□←format∆ts 0 0 0 2  ⍝ no date, time-of-day only
02:00
5
```

The function selects an appropriate format to fit the most demanding timestamp according to the last dimension of the argument, the presence or absence of positive numbers in the year, month and day positions, and of a nonzero fractional part in the seconds position when *6↔¯1↑⍴⍵*. Negative numbers are treated as zeroes. Fractional parts (other than in the sixth position if *6↔¯1↑⍴⍵*) are ignored. If *7≤¯1↑⍴⍵*, the seventh position is treated as milliseconds, and anything that might follow is ignored. The width of the year field in the result is determined by the highest year in the argument.

The possibilities are:

| | |
|---|---|
| *'year-mm-dd'* | *''* |
| *'year-mm-dd hh:mn'* | *'hh:mn'* |
| *'year-mm-dd hh:mn:ss'* | *'mm:mn:ss'* |
| *'year-mm-dd hh:mn:ss.sss'* | *'hh:mn:ss.sss'* |

---

**Title:** *mrab/name∆sort.1*

**Summary:** Returns vector of row indices to sort a matrix of names, telephone directory-style.

**Description:**
Syntax: *r←name∆sort ⍵*

where *⍵* is a 2-dimensional matrix of characters whose rows are names to be sorted, and *r* is a vector of indices such that *⍵[r;]* will be a matrix of alphabetically sorted names.

This function approximates the kind of sorting used in most telephone directories in the following ways:

- hyphens and apostrophes are completely ignored,
- any substring, within a name, that is formed of nonalphanumeric characters (i.e. characters not in *'abc...z01...9'*) is treated as a single space,
- leading definite articles (English and French: the, l', le, la, les) are ignored. (The function can be adjusted for other languages.)

Shortcomings and differences from Bell Canada's algorithm:

- In its present form, the function only supports one font: the regular APL alphabet, not underscored. If multiple-font support is desired, a transliteration of the argument would be preferable to an adjustment of the code.
- Abbreviations and numbers are not spelled out prior to sorting; digits are treated as positions 27 through 36 of the alphabet, and an abbreviation such as *st.* (for *saint*) will be listed between *ss* and *su*.
- The function is likely to blow up on *ws full* if its argument is large relative to unused workspace (i.e. if *⎕wa<3×4 ⎕ws'⍵' ⍴ guesstimate*).

---

**Title:** *pcb/ch∆ravel.1*

**Summary:** Represents an array as character vector, trailing blanks removed.

**Description:**
Represents an array as character vector, trailing blanks removed. The array may be of any rank, with extra CRs to indicate breaks between planes, etc.

---

**Title:** *pcb/datedecode*.1

**Summary:** Returns character array of dates in standard format, from array of daynumbers.

**Description:**
Converts Julian daynumbers to standard character representations, in any order (yy/mm/dd, dd/mm/yy, or mm/dd/yy, as specified by optional left argument). Represents 0 in the argument by blank in the result. Represents the daynumber *past* by the characters '*prior*' and the daynumber *future* by the characters '*current*'.

**Subutilities:**
*pcb/default*.1
*pcb/daterep*.1
*pcb/past*.1
*pcb/future*.1

---

**Title:** *pcb/dateencode*.1

**Summary:** Returns numeric array of Julian daynumbers, from character array of formatted dates.

**Description:**
Calculates Julian daynumbers from standard character forms. the order (dd mm yy, mm dd yy, etc.) expected in the argument is selected by the optional left argument. The function returns daynumber 0 for a date whose representation is blank. It returns the largest representable integer for a date represented by the characters '*current*', and the smallest representable integer (stored as the globals *past* and *future* for a date represented in the argument by the characters '*prior*'. This function is the inverse of *pcb/datedecode*. The numeric part of the calculation is done by *pcb/dno*.

**Subutilities:**
*pcb/dno*.1
*pcb/default*.1
*pcb/notempty*.1
*pcb/past*.1
*pcb/future*.1

---

---

**Title:** *pcb/daterep*.1

**Summary:** Returns 3-element representation of each array of Julian daynumbers.

**Description:**
Converts Julian daynumbers to conventional numeric representations. These are returned in the form yyyy mm dd unless a different order is indicated by the optional left argument. (To convert to characer form with slashes etc, use the function *pcb/datedecode*, which calls this function for the numeric work.)

---

**Title:** *pcb/dno*.1

**Summary:** Returns Julian daynumbers for array of dates. The left argument specifies date format.

**Description:**
Takes array of dates represented as 3-element vectors yyyy mm dd (or other permutation indicated by the optional left argument) and returns an array in which each date is represented by a scalar Julian daynumber. (Note: these are true Julian daynumbers, with origin about 4000 B.C.E.)

---

**Title:** *pcb/fnextract*.1

**Summary:** Extract □*cr* of a function in a package.

**Description:**
Left argument is name of function, right argument package containing that function. Result is □*cr* of the function.

---

**Title:** *pcb/fnname*.1

**Summary:** Extracts function's name from its canonical representation

**Description:**
Scans syntax of header line to pick out the function name.

---

**Title:** *pcb/fnpack.1*

**Summary:** Returns package containing function from □*cr* argument.

**Description:**
Argument is canonical representation of a single function. Result is a package containing that definition as a function (rather than as an array).

**Subutilities:**
*pcb/fnname.1*

---

**Title:** *pcb/matrix.1*

**Summary:** Assures result is a matrix.

**Description:**
Reshape an array argument so that the result is always a matrix, with the same number of columns as the argument, and as many rows as necessary to accomodate all the other axes.

---

**Title:** *pkw/closefx.1*

**Summary:** Closes Epson FX printer.

**Description:**
See *pkw/printfx*.

---

**Title:** *pkw/openfx.1*

**Summary:** Opens Epson FX printer.

**Description:**
Shares variable *prn* with DOSX, and downloads the APL character set. Translation 3 is used. See *pkw/printfx*.

---

---

**Title:** *pkw/printfx.1*

**Summary:** Prints character string on Epson FX printer.

**Description:**
Syntax: *printfx ⍵*

Prints an APL character string with an Epson FX printer that's been opened with *openfx*:

```
openfx ⋄ printfx 1 ⎕fd 'foo' ⋄ closefx
```

**Global variables:**
*fxgraf*
*fxtab*

---

**Title:** *pla/encvecfrmat.1*

**Summary:** Enclosed vector from simple matrix.

**Description:**
*r←encvecfrmat ⍵*

Enclosed vector from simple matrix. Trailing blanks or zeroes are removed from each row before enclosing.

---

**Title:** *rcm/files/compsΔreport.2*

**Summary:** Prepares a report on specified components of a file.

**Description:**
*table←lmt compsΔreport tie*

**Right argument:** integer scalar or 1- or 2-element integer vector containing tie number, and passnumber if necessary.

**Left argument:** first and last components to be reported on. Default setting is all components.

**Result:** character matrix table containing component number, author acct, time written, data type, size in bytes, rank, and shape. For a package, rank and shape are replaced by a list of the names in the package.

**Subutilities:**
*mrab/formatΔts.1*
*wgr/ftt.2*

---

**Title:** *rcm/files/fappend*.1

**Summary:** Cover function for ⎕*append* - resizes file if necessary.

**Description:**
*data fappend tie*

Cover function for ⎕*append* - resizes file if necessary

LEFT ARGUMENT - data item to be appended

RIGHT ARGUMENT - integer scalar or 1 or 2 element integer vector - file tie number, and passnumber if necessary

**Note:** If file reservation is exceeded, program will error out, since this error requires user intervention to resolve.

---

**Title:** *rcm/files/fappendr*.1

**Summary:** Cover function for ⎕*appendr* - resizes file if necessary.

**Description:**
*pointer←data fappendr tie*

Cover function for ⎕*appendr* - resizes file if necessary

LEFT ARGUMENT - data item to be appended

RIGHT ARGUMENT - integer scalar or 1 or 2 element integer vector - file tie number, and passnumber if necessary

NOTE: if file reservation is exceeded, program will error out, since this error requires user intervention to resolve.

---

**Title:** *rcm/files/fcompare.2*

**Summary:** Compares two files that have the same number of components.

**Description:**
*lfile fcompare rfile*

Compares two files which have the same number of components, some of which may have been replaced with different values.

**Right argument:** file tie number of one file

**Left argument:** file tie number of the other file

The result is displayed, not returned explicitly. It details any differences between corresponding components, stating if one is a package and the other is not, or if they are both arrays differing in type, shape, or data elements, or if they are differing packages.

**Subutilities:**
*mgf/pkgmatch.1*
*pcb/fnextract.1*

---

**Title:** *rcm/files/fcompress.1*

**Summary:** Analog to primitive / for files.

**Description:**
*outfile←control fcompress infile*

Analog to primitive / for files.

RIGHT ARGUMENT - input file tie number

LEFT ARGUMENT - boolean vector specifying components to keep or delete

RESULT - output file tie number, of the newly created file

---

**Title:** *rcm/files/fcopy.1*

**Summary:** Copies part or all of file as specified.

**Description:**
*outfile←outfile fcopy infile*

Copies part or all of file as specified

RIGHT ARGUMENT - integer vector of 1-5 elements: [1] input file tie number, [2] first component to be copied, [3] last component to be copied, [4] component increment, [5] input file passnumber

LEFT ARGUMENT - tie number or name of output file

RESULT - tie number of output file

---

**Title:** *rcm/files/fdrop.1*

**Summary:** Analog to primitive ↓ for files.

**Description:**
*outfile←control fdrop infile*

Analog to primitive ↓ for files

RIGHT ARGUMENT - input file tie number

LEFT ARGUMENT - integer specifying number of components to drop

RESULT - output file tie number, of the newly created file

---

**Title:** *rcm/files/ferase.1*

**Summary:** Erases a file specified by argument.

**Description:**
*ferase file*

Erases a file specified by argument.

ARGUMENT - integer scalar tie number OR char vector file name

---

---

**Title:** *rcm/files/fexpand.1*

**Summary:** Analog to primitive \ for files.

**Description:**
*outfile←control fexpand infile*

Analog to primitive \ for files.

RIGHT ARGUMENT - input file tie number

LEFT ARGUMENT - boolean vector specifying where to add or copy components

RESULT - output file tie number, of the newly created file

---

**Title:** *rcm/files/fextent.3*

**Summary:** Returns type, space, rank, and shape of specified components.

**Description:**
*extent←lmt fextent tie*

Returns type, space, rank, and shape of specified components.

RIGHT ARGUMENT - integer scalar or 1 or 2 element integer vector. file tie number, and passnumber if necessary

LEFT ARGUMENT - 2 element integer vector. Lowest and highest component numbers for which extent is to be obtained. Default is all components.

RESULT - type (0 for package, 1 for boolean, 2 for character, 3 for integer, 4 for floating-point, 5 for complex, 6 for enclosed), bytes used, rank, shape (padded with 0's if necessary)

---

---

**Title:** *rcm/files/findex.1*

**Summary:** Analog to primitive [] for files.

**Description:**
*outfile←control findex infile*

Analog to primitive [] for files.

RIGHT ARGUMENT - input file tie number

LEFT ARGUMENT - integer vector specifying components to select

RESULT - output file tie number, of the newly created file

This function works only in origin 1.

---

**Title:** *rcm/files/finsert.1*

**Summary:** Inserts data item into copy of input file at specified location.

**Description:**
*outfile←data finsert tie cn*

**Right argument:** 2- or 3-element numeric vector: tie number, component number after which data is to be inserted, and passnumber if necessary

**Left argument:** data value to be inserted in file

**Result:** file tie number of copy of the newly created file file with data inserted

---

**Title:** *rcm/files/flib.1*

**Summary:** Cover function for □*lib* - returns alphabetized file library list.

**Description:**
*lib←flib acct*

Cover function for □*lib* - returns alphabetized file library list.

ARGUMENT - integer scalar or 1 element vector account number

RESULT - n×22 character matrix of file names

---

---

**Title:** *rcm/files/fmatch.1*

**Summary:** Matches components of two files where one file has had components inserted or the other has had components deleted.

**Description:**
*lfile fmatch rfile*

Matches components of two files where the left file has had components inserted or the right file has had components deleted

**Right argument:** tie number of file with fewer components

**Left argument:** tie number of file with more components

The result is displayed, not returned explicitly. It is similar to *leftfile* ∘.≡ *rightfile*.

**Subutilities:**
*mgf/pkgmatch.1*
*pcb/fnextract.1*

---

**Title:** *rcm/files/fopen.2*

**Summary:** Share ties a file, and creates it if necessary.

**Description:**
*tie←size fopen name*

**Right argument:** character vector name of file

**Left argument optional:** bytes to reserve for file. If specified, file will be cleared of all existing components

**Result:** integer scalar specifying number file was tied to

**Subutilities:**
*rohan/make∆std∆filename.1*

---

---

**Title:** *rcm/files/fprint.1*

**Summary:** Prints characteristics and/or values of components in a file.

**Description:**
*control fprint file*

Prints characteristics and values of components in a file.

**Right argument:** integer scalar file tie number to be processed

**Left argument:** a four-element boolean vector: [1] controls printing of value [2] controls printing of component number [3] control printing data type [4] controls printing shape.

---

**Title:** *rcm/files/freplace.1*

**Summary:** Cover function for ⎕*replace* - resizes file if necessary.

**Description:**
*data freplace tie*

Cover function for ⎕*replace* - resizes file if necessary.

RIGHT ARGUMENT - 2 or 3 element integer vector - file tie number, component number, and if necessary, passnumber

LEFT ARGUMENT - data item to be write over existing component

NOTE - If file reservation is exceeded, program will error out, since this error requires user intervention to resolve.

---

**Title:** *rcm/files/ftake.1*

**Summary:** Analog to primitive ↑ for files.

**Description:**
*outfile←control ftake infile*

Analog to primitive ↑ for files.

RIGHT ARGUMENT - input file tie number

LEFT ARGUMENT - integer specifying number of components to take

RESULT - output file tie number, of the newly created file

The left argument may be either positive or negative. Overtaking is NOT done--the function will halt on a file index error if |*control* is too large.

---

**Title:** *rohan/centre.1*

**Summary:** Centres a vector of text (such as a heading for a report).

**Description:**

*result*←[*width*] *centre vector*:

Centres the given *vector* of text in a field of the specified *width*, by catenating leading and trailing blanks as required. The given vector should not already have any leading or trailing blanks, nor any embedded backspaces. The default width is □*pw*.

If the given vector is already longer than the width will allow, a domain error will occur within the function.

**Title:** *rohan/columnize.2*

**Summary:** Given a namelist (or similar matrix), puts the names in columns across the page.

**Description:**
*result*←*width columnize matrix* (*width* is optional): *matrix* is a matrix of characters, with each row containing one name or other unit of information. *result* has the matrix split up into columns which are placed side by side into a new matrix of width *width* (the default width is ⎕*pw*), with at least one blank between columns. Requires *width*≥¯1↑⍴*matrix* (otherwise a domain error will occur inside the function).

Examples:

```
      22 columnize m←11 6⍴'mary
                          had
                          a
                          little
                          lamb
                          whose
                          fleece
                          was
                          white
                          as
                          snow  '
mary   lamb   white
had    whose  as
a      fleece snow
little was
      32 columnize m
mary    little  fleece  as
had     lamb    was     snow
a       whose   white
```

The number of columns is ⌊(*width*+1)÷1+¯1↑⍴*matrix*; the number of rows is ⌈(1↑⍴*matrix*)÷*numberΔofΔcolumns*. ⍴*result* ↔ *numberΔofΔrows*,*width*.

---

**Title:** *rohan/day.2*

**Summary:** Returns the day of the week (*monday*, *tuesday*, etc.) that a date falls on.

**Description:**
*z*←*day date*: returns the day of the week (*monday*, *tuesday*, etc.) that the given date falls on. The date must be in 3↑⎕*ts* format, e.g. 1982 11 21. The result is a vector of characters.

---

---

Title: *rohan/easytie.2*

Summary: Ties a file and returns the tie number, with provision for later untying a newly tied file.

Description:
Syntax: *tienum←easytie filename*. The filename must be in the usual form required by ⎕*tie* and ⎕*stie*.

*easytie* ties the specified file to the smallest number not already in use, and returns this tie number as a scalar. If the file was already tied, then the function just returns the tie number as a single-element vector.

This function is intended for use with *rohan/easyuntie*, which unties a file only if *easytie* actually had to tie it (and not simply return an existing tie number). Thus, *easyuntie* restores the tie state of a file to what it had been before *easytie* was executed. For example:

```
    ∇ foo;ctie;utie
[1]   ctie←easytie '1234567 catalogue'
[2]   utie←easytie 'updates'
[3]   (⎕read utie,1) ⎕replace ctie,1
[4]   easyuntie ctie
[5]   easyuntie utie
[6]   ⍝ note that easyuntie ctie,utie will not work!
    ∇
```

*foo* leaves ⎕*nums* and ⎕*names* as they were just before it was executed.

*rohan/easytie* operates correctly with either value of Lio.

Subutilities:
*rohan/easyuntie.1*

---

Title: *rohan/easyuntie.1*

Summary: Unties a file tied by *rohan/easytie*.

Description:
Unties a file tied by *rohan/easytie*. See the description of *rohan/easytie* for details.

---

---

**Title:** *rohan/filespace.1*

**Summary:** Returns names and □*sizes* of all files in the specified library, in a character matrix with total.

**Description:**
*result←filespace library*: returns the names and □*sizes* of all files in the specified library, in a character matrix with total. Uses zeros for □*sizes* of files which require passnumber access.

Side effect: Unties all tied files.

---

**Title:** *rohan/format∆nl.1*

**Summary:** Formats a matrix namelist just like )*fns*, )*vars*, and )*grps* do.

**Description:**
*z←width format∆nl namelist* (*width* is optional): Formats a matrix namelist just like )*fns*, )*vars*, and )*grps* do. The result is an n by *width* matrix; the default width is □*pw*.

---

**Title:** *rohan/format∆timestamp.1*

**Summary:** Puts the given □*ts*-style timestamp into the format *hh.mm.ss www dd mmm yyyy*.

**Description:**
*result←format∆timestamp ts*: puts the given □*ts*-style timestamp into the format *hh.mm.ss www dd mmm yyyy* (the same format used by the SHARP APL Message Processing Facility, MAILBOX).

Example:

```
      format∆timestamp □←□ts
1982 12 9 2 45 46 136
 2.45.46  thu  9 dec 1982
```

---

---

Title: *rohan/if.1*

Summary: *→if condition ◇ action∆if∆condition∆true*

Description:
Usage: *→if condition ◇ action∆if∆condition∆true*

For example:

```
    ∇ test
[1]   ⍝ this line is always executed
[2]   →if (lts ⍕ts)[4]<7 ◇ 'insomnia, eh?' ◇ 'try counting s
heep.'
[3]   ⍝ this line is always executed
    ∇
```

Note that (as in line [2] above) the action to be taken can occupy more than one statement.

There is also a complementary utility *rohan/unless*, which is used as follows:

*→unless condition ◇ action∆unless∆condition∆true*

---

Title: *rohan/make∆nl.1*

Summary: Takes a character-array argument of names, of any rank, and returns a left-justified matrix namelist.

Description:
*result←make∆nl array*: takes a character-array argument of names, of any rank, and returns a left-justified matrix namelist. For example:

```
      make∆nl '    fnshow   seshow by   and    '
fnshow
seshow
by
and
```

---

---

**Title:** *rohan/makeΔstdΔfilename.1*

**Summary:** Puts a filename into the standard 22-element form returned by ⎕*lib and* ⎕*names*.

**Description:**
*result←makeΔstdΔfilename a*: puts the given filename (which may or may not contain a library number) into the standard 22-element form returned by ⎕*lib and* ⎕*names*. The argument must be a vector (or scalar) of characters; the result is a 22-element vector of characters.

Examples:

```
      '→',(makeΔstdΔfilename '444 mailsystem'),'←'
→       444 mailsystem ←

      '→',(makeΔstdΔfilename '444      mailsystem'),'←'
→       444 mailsystem ←

      '→',(makeΔstdΔfilename 'myfile'),'←'
                                Assuming that 1↑⎕ai is 1234567
→   1234567 myfile      ←
```

---

**Title:** *rohan/simple.2*

**Summary:** Determines whether an array is simple.

**Description:**
*result←simple array*: determines whether the given array is simple (i.e. does not contain any enclosures). Note that by definition an empty array is always simple. For instance:

```
      simple 3 4⍴⍳12
1
      simple ⎕av
1
      simple x←(<'abcd'),<2 2⍴3.4
0
      simple 0⍴x
1
```

---

---

**Title:** *rohan/sizes.2*

**Summary:** Returns a formatted table of the names and sizes of the objects specified in the argument.

**Description:**
*z←width sizes namelist* (*width* is optional): the right argument is a matrix namelist of objects in the active workspace; the result is a formatted matrix of the objects' names and sizes, in decreasing order of size. The optional left argument specifies the maximum width of the result; the default value is *⎕pw*.

You may find this function very useful when you are faced with *ws full* problems. It will give you some idea of where the space in your workspace is going.

---

**Title:** *rohan/subtotal.1*

**Summary:** Inserts first-axis subtotals into an array.

**Description:**
*result←boolean subtotal array*: returns *array* with first-axis subtotals inserted. Each 1 in vector *boolean* designates the beginning of a new section for subtotalling.

Example:

```
      1 0 1 1 0 0 0 subtotal 7 3 ⍴⍳21
1  2  3
4  5  6
5  7  9
7  8  9
7  8  9
10 11 12
13 14 15
16 17 18
19 20 21
58 62 66
```

*⍴boolean ↔ 1↑⍴array*; *1↑⍴result ↔ (⍴boolean)++/boolean*. Note that *boolean* must (unless empty) begin with a 1.

**Subutilities:**
*sauce/plengths.2*
*sauce/pplusreduce.1*

---

---

**Title:** *rohan/tied.1*

**Summary:** Shows which files are tied to what numbers, in a readable format.

**Description:**
Shows which files are tied to what numbers, in a readable format. For example:

```
      tied
        private      1
     86 mabrav3      100
        msk99        101
1854339 stuff        12
      ptied
4 33
```

---

**Title:** *rohan/unless.1*

**Summary:** *→unless condition ◇ action∆unless∆condition∆true.*

**Description:**
Usage is: *→unless condition ◇ action∆unless∆condition∆true*

For example:

```
    ∇ test
[1]   ⍝ this line is always executed
[2]   →unless ⎕wa≥5000 ◇ 'insufficient working space' ◇ →0
[3]   ⍝ this line is executed if and only if ⎕wa≥5000
    ∇
```

Note that (as in line [2] above) the action to be taken can occupy more than one statement.

There is also a complementary utility *rohan/if*, which is used as follows:

*→if condition ◇ action∆if∆condition∆true*

---

**Title:** *sauce/alloceq.1*

**Summary:** Allocates a number equitably subject to limits. E.g. 29 *alloceq* 3 20 500 7 ↔ 3 9.5 9.5 7.

**Description:**
*result←total alloceq limits*

*total* is a singleton and *limits* is a vector. The function allocates *total* equitably subject to *limits*.

E.g. 29 *alloceq* 3 20 500 7 ↔ 3 9.5 9.5 7.

---

---

**Title:** *sauce/allocfifo*.1

**Summary:** Allocates a number on a FIFO basis, subject to limits. E.g. `9 allocfifo 4 6 11 ↔ 4 5 0.`

**Description:**
*result←total allocfifo limits*

*total* is a singleton and *limits* is a vector. The function allocates *total* on a first-in first-out basis subject to *limits*.

E.g. `9 allocfifo 4 6 11 ↔ 4 5 0.`

---

**Title:** *sauce/catenaterows*.1

**Summary:** Catenates one vector or matrix under another, using overtake as necessary.

**Description:**
*matrix←top catenaterows bottom*

Catenates the *top* row(s) and the bottom row(s), which must both be of rank at most 2.

Examples:

```
      ⍴⍞←'hi' catenaterows 'there'
hi
there
2 5
      ⍴⍞←(2 3⍴'abcdef') catenaterows 'gh'
abc
def
gh
3 3
      ⍴⍞←(2 3⍴'abcdef') catenaterows 3 4⍴'wxyz'
abc
def
wxyz
wxyz
wxyz
5 4

      (2 1 ⍴ 4 3)≡4 catenaterows 3
1
```

---

---

**Title:** *sauce/cyclicmesh.1*

**Summary:** E.g. 4 *cyclicmesh* 2 12⍴'*aei*','*bfj*','*cgk*','*dhl*' ↔ 2 12⍴'*abcdefghijkl*'.

**Description:**
*result←n cyclicmesh array*

E.g. 4 *cyclicmesh* 2 12⍴'*aei*','*bfj*','*cgk*','*dhl*' ↔
2 12⍴'*abcdefghijkl*'

*array* may be of arbitrary rank; it may contain numbers, characters, or enclosures.

---

**Title:** *sauce/distinct.1*

**Summary:** Removes duplicate elements from a vector; e.g. *distinct* 3 5 3 4 ↔ 3 5 4.

**Description:**
*compressed←distinct vector*

Removes duplicate elements from *vector*. *vector* must have rank ≤1; it may contain characters, numbers, or enclosures.

Examples:

```
      distinct 3 5 3 4
3 5 4
      distinct 'abbbcbbd'
'abcd'
```

---

**Title:** *sauce/distinctrows.1*

**Summary:** Removes duplicate rows from matrix; e.g. *distinctrows* 4 2⍴'*ab*','*cd*','*ab*','*ad*' ↔ 3 2⍴'*ab*','*cd*','*ad*'.

**Description:**
*compressed←distinctrows matrix*

Removes duplicate rows from *matrix*. *matrix* may contain characters or numbers; it should not contain enclosures.

Example:

```
      distinctrows 6 2 ⍴'ab','ab','cd','ab','cd','ef'
ab
cd
ef
```

---

**Title:** *sauce/distround.1*

**Summary:** Distributive rounding; e.g. 0.01 *distround* 2 6 ⍴÷ 1 3 ↔ 2 6 ⍴ 1 0.34 1 0.33 1 0.33.

**Description:**
*rounded←unit distround array*

Performs rounding ensuring that *+/array* ↔ *+/rounded*, when both are expressed in the same *units*. Subject to this, *+/|array-rounded* is minimized.

E.g. 0.01 *distround* 2 6 ⍴÷ 1 3 ↔ 2 6 ⍴ 1 0.34 1 0.33 1 0.33.

E.g. 0.01 *distround* 9⍴÷3 ↔ 0.34 0.34 0.34 0.33 0.33 0.33 0.33 0.33 0.33.

E.g. 0.01 *distround* 2 2 ⍴÷3 ↔ 2 2 ⍴ 0.34 0.33 0.34 0.33.

---

**Title:** *sauce/elemreplace.1*

**Summary:** Replaces, in a vector containing no enclosures, all occurrences of a scalar by a (different) vector.

**Description:**
*result←vector elemreplace elemstring*

Replaces, in *vector*, all occurrences of *1↑elemstring* by *1↓elemstring*. *vector* may be character or numeric; it should not contain enclosures.

E.g. *'abaacda' elemreplace 'a','zap'* ↔ *'zapbzapzapcdzap'*.

E.g. *'abaacda' elemreplace 'a'* ↔ *'bcd'*.

E.g. *'abaacda' elemreplace ''* ↔ *'abaacda'*.

E.g. *' b cd ' elemreplace ''* ↔ *'bcd'*.

---

**Title:** *sauce/extendparts*.1

**Summary:** For extending parts; e.g. 1 1 0 0 1 0 0 *extendparts* 1 0 0 0 1 1 0 ↔ 1 0 0 0 1 0 0 1 0 0 0.

**Description:**
*mask←selector extendparts boolean*

Each 1 in *boolean* designates the beginning of a part. The result is an expansion mask that can be used to extend parts. *selector* determines how much each part is to be expanded.

For example:

```
      1 1 0 0 1 0 0 extendparts 1 0 0 0 1 1 0
1 0 0 0 1 0 0 1 0 0 0.

      1 1 0 0 1 0 0 extendparts 1 1 0 1
1 1 0 0 0 1 0 0.
```

---

**Title:** *sauce/firstones*.1

**Summary:** Sets to 0 all but the first 1 in each sequence of 1's in a boolean vector.

**Description:**
*result←firstones boolean*

*boolean* must be vector or scalar. ⍴*result* ↔ ⍴,*boolean*, with all but the first 1 in each sequence of 1's set to 0.

```
E.g. firstones  1 0 1 1 1 0 1 1 0 0 1 1  ↔
1 0 1 0 0 0 1 0 0 0 1 0.
```

---

**Title:** *sauce/firstzeroes*.1

**Summary:** Sets to 1 all but the first 0 in each sequence of 0's in a boolean vector.

**Description:**
*result←firstzeroes boolean*

*boolean* must be vector or scalar. ⍴*result* ↔ ⍴,*boolean*, with all but the first 0 in each sequence of 0's set to 1.

```
E.g. firstzeroes  0 1 0 0 0 1 0 0 1 1 0 0  ↔
0 1 0 1 1 1 0 1 1 1 0 1.
```

---

---

Title: *sauce/from*.1

**Summary:** A variant of indexing. Each row of the left argument selects one element from the right argument.

**Description:**
*elements←indices from array*

A variant of indexing. Each row of *indices* selects one element from *array*. ⍴*indices* ↔ (⍴*elements*),⍴⍴*array*.

E.g. [⎕*io*←1] (3 2 ⍴ 1 4 , 2 1 , 1 3) *from* 2 4 ⍴'*abcd*','*efgh*'
↔ '*dec*'.

E.g. ⍴(? 3 2 7 5 ⍴6) *from* 10 9 8 7 6 ⍴2 ↔ 3 2 7.

---

Title: *sauce/fromto*.1

**Summary:** E.g. 1 157 ¯12 *fromto* 2 161 ¯10 ↔ 1 2 157 158 159 160 161 ¯12 ¯11 ¯10.

**Description:**
*result←origins fromto ends*

E.g. 1 157 ¯12 *fromto* 2 161 ¯10 ↔
1 2 157 158 159 160 161 ¯12 ¯11 ¯10.

---

**Title:** *sauce/gerows.1*

**Summary:** Determines which rows of a matrix are ≥ (in the sense of ⍋) a vector.

**Description:**
*boolean←matrix gerows vector*

The result is a boolean vector of length 1↑⍴*matrix*. *boolean*[*i*]=1 if and only if *matrix*[*i*;] is greater than or equal to *vector* (in the sense of ⍋).

Examples

```
      mat
2 7 5
5 2 1
2 7 9
4 5 8
      mat gerows 2 7 7
0 1 1 1
      mat gerows 2 8 ¯1.5
0 1 0 1
      mat gerows 2 7 9
0 1 1 1
```

---

**Title:** *sauce/gradedown.1*

**Summary:** Applies ⍒ over the designated axis of an array.

**Description:**
*indices←[axis] gradedown array*

⍒ is applied over the designated *axis* of *array*. The default *axis* is the last axis of *array*.

E.g.

```
      ⎕io←1
      gradedown 1 3 3 ⍴ 1 3 2 , 6 5 4 , 8 9 7
  2 3 1
  1 2 3
  2 1 3
```

---

**Title:** *sauce/gradeup*.1

**Summary:** Applies ⍋ over the designated axis of an array.

**Description:**
*indices*←[*axis*] *gradeup array*

⍋ is applied over the designated *axis* of *array*. The default *axis* is the last axis of *array*.

E.g.
```
      ⎕io←1
      gradeup 1 3 3 ⍴ 1 3 2 , 6 5 4 , 8 9 7
1 3 2
3 2 1
3 1 2
```

---

**Title:** *sauce/gtrows*.1

**Summary:** Determines which rows of a matrix are > (in the sense of ⍋) a vector.

**Description:**
*boolean*←*matrix gtrows vector*

The result is a boolean vector of length 1↑⍴*matrix*. *boolean*[*i*]=1 if and only if *matrix*[*i*;] is greater than *vector* (in the sense of ⍋).

Examples:
```
      mat
2 7 5
5 2 1
2 7 9
4 5 8
      mat gtrows 2 7 7
0 1 1 1
      mat gtrows 2 8 ¯1.5
0 1 0 1
      mat gtrows 2 7 9
0 1 0 1
```

---

**Title:** *sauce/indexgen.1*

**Summary:** The argument should be a vector. Returns (⍳1↑*vector*),(⍳1↑1↓*vector*),(⍳1↑2↓*vector*),..., ⍳¯1↑*vector*.

**Description:**
*result*←*indexgen vector*

*result*←(⍳1↑*vector*),(⍳1↑1↓*vector*),(1↑2↓*vector*),...,⍳¯1↑*vector*

---

**Title:** *sauce/indexofrows.2*

**Summary:** Returns the 'index of' each row of one matrix in another matrix.

**Description:**
*indices*←*matrix indexofrows rows*

*rows* may be a vector or matrix of any shape. *indices* contains the 'index of' each row of *rows* in *matrix*. ⍴*indices* ↔ ¯1↓⍴*rows*.

Examples:

```
      ⎕io←1
      (4 2 ⍴'ab','cd','ef','gh') indexofrows 3 2 ⍴'ef','ab',
'yz'
3 1 5
      ⎕io←0
      (4 2 ⍴'ab','cd','ef','gh') indexofrows 'cd'
1
```

Both *rows* and *matrix* may contain characters, numbers, or enclosures. If either *rows* or *matrix* contains enclosures, (¯1↑⍴*matrix*) ↔ ¯1↑1,⍴*rows*.

---

**Title:** *sauce/lastones.1*

**Summary:** Sets to 0 all but the last 1 in each sequence of 1's in a boolean vector.

**Description:**
*result*←*lastones boolean*

*boolean* must be vector or scalar. ⍴*result* ↔ ⍴,*boolean*, with all but the last 1 in each sequence of 1's set to 0.

E.g. *lastones* 1 0 1 1 1 0 1 1 0 0 1 1 ↔ 1 0 0 0 1 0 0 1 0 0 0 1.

---

---

**Title:** *sauce/lastzeroes.1*

**Summary:** Sets to 1 all but the last 0 in each sequence of 0's in a boolean vector.

**Description:**
*result←lastzeroes boolean*

*boolean* must be vector or scalar. ⍴*result* ↔ ⍴,*boolean*, with all but the last 0 in each sequence of 0's set to 1.

E.g. *lastzeroes* 0 1 0 0 0 1 0 0 1 1 0 0 ↔ 0 1 1 1 0 1 1 0 1 1 1 0.

---

**Title:** *sauce/leftjustify.1*

**Summary:** Left justifies each row of an array.

**Description:**
*result←[element] leftjustify array*

The default *element* is 1↑0⍴*array* (i.e. 0 or ''). The function left justifies each row of *array* with respect to *element*. Note: it will work even if *element* has more that one distinct element; in this case, left justification is done with respect to all the elements of *element*.

```
E.g.  leftjustify '  joe     ↔  'joe
                  mary          mary
                   john          john
                     ed'         ed  '
```

```
∈.g.  ' ∘' leftjustify '∘∘∘joe    ↔  'joe∘∘∘
                       mary∘∘         mary∘∘
                        john∘         john∘
                       ∘  ed∘'        ed∘∘  '
```

*array* may contain characters, numbers, or enclosures.

---

**Title:** *sauce/maskparts.1*

**Summary:** E.g. `1 0 1 0 maskparts 1 0 0 1 0 1 0 1 0 ↔ 1 1 1 0 0 1 1 0 0.`

**Description:**
*result←mask maskparts boolean*

Each 1 in *boolean* designates the beginning of a part. The function masks each part according to the boolean *mask*.

E.g. `1 0 1 0 maskparts 1 0 0 0 1 0 1 0 1 0 ↔ 1 1 1 0 0 1 1 0 0.`

---

**Title:** *sauce/memberrows.1*

**Summary:** Returns a boolean array indicating whether each row of one argument is a row of the other arg.

**Description:**
*boolean←rows memberrows matrix*

*rows* may be an array of any shape. The result is a boolean array of shape `¯1↓⍴rows` indicating which rows of *rows* are rows of *matrix*. Neither *rows* nor *matrix* may contain enclosures.

Example:

```
      'ab  memberrows 'xyz   ↔ 1 0
       xy'             ab
                       cde'
```

---

**Title:** *sauce/mesh*.1

**Summary:** Meshes catenated arrays; e.g. 1 2 3 3 2 1 *mesh* 2 6 ⍴'*ad*','*be*','*cf*' ↔ 2 6 ⍴'*abcdef*'.

**Description:**
*result*←*indices mesh array*

Meshes catenated arrays.
E.g. 1 2 3 3 2 1 *mesh* 2 6 ⍴'*ad*','*be*','*cf*' ↔
2 6 ⍴'*abcfed*'.

It is assumed that ¯1↑1,⍴*array* ↔ ⍴,*indices*.

E.g. 1 2 2 1 1 2 *mesh* '*adb*','*ecf*' ↔ '*aecdbf*'.

E.g. 1 3 1 3 2 1 3 3 *mesh* '*abc*','*m*','*wxyz*' ↔ '*awbxmcyz*'.

*array* may contain characters, numbers, or enclosures.

---

**Title:** *sauce/notmemberrows*.1

**Summary:** Returns a boolean array indicating whether each row of one argument is not a row of the other argument.

**Description:**
*boolean*←*rows memberrows matrix*

*rows* may be an array of any shape. The result is a boolean array of shape ¯1↓⍴*rows* indicating which rows of *rows* are not rows of *matrix*. Neither *rows* nor *matrix* may contain enclosures.

Example:

```
      'ab  notmemberrows 'xyz    ↔  0 1
       xy'               ab
                         cde'
```

---

**Title:** *sauce/pandreduce*.1

**Summary:** Partitioned ∧/*array*; each 1 in the boolean left argument designates the beginning of a part in the array.

**Description:**
*result*←*boolean pandreduce array*

Partitioned ∧/*array*. *boolean* is a vector with ⍴*boolean* ↔ ¯1↑⍴*array* (although scalar extension may be performed). Each 1 in *boolean* designates the beginning of a part of *array*.

Examples:

```
      1 0 0 0 1 0 1 0 pandreduce 1 1 0 1 1 1 0 0
  ↔  (∧/ 1 1 0 1),(∧/ 1 1),(∧/ 0 0)

      1 0 1 0 0 0 pandreduce 2 6 ⍴0,11⍴1
0 1
1 1
```

It is assumed that ∧/,*array*∈ 0 1 ↔ 1.

---

**Title:** *sauce/pandscan*.1

**Summary:** Partitioned ∧*array*; each 1 in the boolean left argument designates the beginning of a part in the array.

**Description:**
*result*←*boolean pandscan array*

Partitioned ∧*array*. *boolean* is a vector with ⍴*boolean* ↔ ¯1↑⍴*array* (although scalar extension may be performed). Each 1 in *boolean* designates the beginning of a part of *array*.

Example:

```
      1 0 0 0 1 0 1 0 pandscan 1 1 0 1 1 1 0 0 1
  ↔  (∧\ 1 1 0 1),(∧\ 1 1),(∧\ 0 0)


      1 0 1 0 0 0 pandscan 2 6 ⍴ 1 0 1 0 1 0 , 1 1 0 1 1 1
1 0 1 0 0 0
1 1 0 0 0 0
```

It is assumed that ∧/,*array*∈ 0 1 ↔ 1.

---

**Title:** *sauce/pegscan.1*

**Summary:** Partitioned =*array*; each 1 in the boolean left argument designates the beginning of a part in the array.

**Description:**
*result←boolean pegscan array*

Partitioned =*array*. *boolean* is a vector with ⍴*boolean* ↔ ¯1↑⍴*array* (although scalar extension may be performed). Each 1 in *boolean* designates the beginning of a part of *array*.

Examples:

```
      1 0 0 0 1 0 1 0 pegscan 1 1 0 1 1 0 0 1
  ↔   (=\ 1 1 0 1),(=\ 1 0),(=\ 0 1)

      1 0 1 0 0 0 pegscan 2 6 ⍴ 1 0 1 0 1 0 , 1 1 0 1 1 1
1 0 1 0 0 1
1 1 0 0 0 0
```

It is assumed that ∧/,*array*∈ 0 1 ↔ 1.

---

**Title:** *sauce/pgradedown.1*

**Summary:** Partitioned ⍒ over the last axis of of an array.

**Description:**
*result←boolean pgradedown array*

Partitioned ⍒ over the last axis of *array*. *boolean* is a vector with ⍴*boolean* ↔ ¯1↑⍴*array* (although scalar extension may be performed). Each 1 in *boolean* designates the beginning of a part of *array*.

Examples:

```
      1 0 0 0 1 0 1 0 pgradedown 2 5 3 4 9 8 0.3 ¯1
  ↔   (⍒ 2 5 3 4),(⍒ 9 8),(⍒ 0.3 ¯1)

      1 0 1 0 0 0 pgradedown 2 6 ⍴ 1 3 2 6 4 8 , 0.2 12 ¯1 ¯
2 ¯3 ¯4
  ↔   2 6 ⍴⍳⎕io+ 1 0 3 1 2 0 , 1 0 3 2 1 0
```

---

**Title:** *sauce/pgradeup.1*

**Summary:** Partitioned ⍋ over the last axis of an array.

**Description:**
*result←boolean pgradeup array*

Partitioned ⍋ over the last axis of *array*. *boolean* is a vector with ⍴*boolean* ↔ ¯1↑⍴*array* (although scalar extension may be performed). Each 1 in *boolean* designates the beginning of a part of *array*.

Examples:

```
      1 0 0 0 1 0 1 0 pgradeup 2 5 3 4 9 8 0.3 ¯1
  ↔  (⍋ 2 5 3 4),(⍋ 9 8),(⍋ 0.3 ¯1)

      1 0 1 0 0 0 pgradeup 2 6 ⍴ 1 3 2 6 4 8 , 0.2 12 ¯1 ¯2
¯3 ¯4
  ↔  2 6 ⍴⎕io+ 0 1 0 2 1 3 , 0 1 0 1 2 3
```

---

**Title:** *sauce/plengths.2*

**Summary:** E.g. *plengths* 1 1 0 0 1 0 0 0 0 1 0 ↔ 1 3 5 2.

**Description:**
*lengths←plengths boolean*

E.g. *plengths* 1 1 0 0 1 0 0 0 0 1 0 ↔ 1 3 5 2.

---

**Title:** *sauce/plescan.1*

**Summary:** Partitioned ≤*array*; each 1 in the boolean left argument designates the beginning of a part in the array.

**Description:**
*result←boolean plescan array*

Partitioned ≤*array*. *boolean* is a vector with ⍴*boolean* ↔ ¯1↑⍴*array* (although scalar extension may be performed). Each 1 in *boolean* designates the beginning of a part of *array*.

Examples:

```
  1 0 0 0 1 0 1 0 plescan 1 1 0 1 1 0 0 1
  ↔  (≤\ 1 1 0 1),(≤\ 1 0),(≤\ 0 1)

      1 0 1 0 0 0 plescan 2 6 ⍴ 1 0 1 0 1 0 , 1 1 0 1 1 1
1 0 1 0 1 1
1 1 0 1 1 1
```

It is assumed that ∧/,*array*∈ 0 1 ↔ 1.

---

---

**Title:** *sauce/pltscan.1*

**Summary:** Partitioned *\array*; each 1 in the boolean left argument designates the beginning of a part in the array.

**Description:**
*result←boolean pltscan array*

Partitioned *\array*. *boolean* is a vector with ⍴*boolean* ↔ ¯1↑⍴*array* (although scalar extension may be performed). Each 1 in *boolean* designates the beginning of a part of *array*.

Examples:

```
      1 0 0 0 1 0 1 0 pltscan 1 1 0 1 1 0 0 1
  ↔  (<\ 1 1 0 1),(<\ 1 0),(<\ 0 1)

      1 0 1 0 0 0 pltscan 2 6 ⍴ 1 0 1 0 1 0 , 1 1 0 1 1 1
1 0 1 0 0 0
1 0 0 1 0 0
```

It is assumed that ∧/,*array*∈ 0 1 ↔ 1.

---

**Title:** *sauce/pmaxreduce.1*

**Summary:** Partitioned ⌈*/array*; each 1 in the boolean left argument designates the beginning of a part in the array.

**Description:**
*result←boolean pmaxreduce array*

Partitioned ⌈*/array*. *boolean* is a vector with ⍴*boolean* ↔ ¯1↑⍴*array* (although scalar extension may be performed). Each 1 in *boolean* designates the beginning of a part of *array*.

Examples:

```
      1 0 0 0 1 0 1 0 pmaxreduce ¯1 2 8 3 9 0.5 12 14
  ↔  (⌈/ ¯1 2 8 3),(⌈/ 9 0.5),(⌈/ 12 14)


      1 0 1 0 0 0 pmaxreduce 2 6 ⍴ 1 2 6 4 5 3 , 0.5 ¯3 ¯5 ¯
4 ¯6 ¯6
2    6
0.5 ¯4
```

---

**Title:** *sauce/pmaxscan.1*

**Summary:** Partitioned ⌈*array*; each 1 in the boolean left argument designates the beginning of a part in the array.

**Description:**
*result←boolean pmaxscan array*

Partitioned ⌈*array*. *boolean* is a vector with ⍴*boolean* ↔ ¯1↑⍴*array* (although scalar extension may be performed). Each 1 in *boolean* designates the beginning of a part of *array*.

Examples:

```
      1 0 0 0 1 0 1 0 pmaxscan ¯1 2 8 3 9 0.5 12 14
  ↔  (⌈\ ¯1 2 8 3),(⌈\ 9 0.5),(⌈\ 12 14)


      1 0 1 0 0 0 pmaxscan 2 6 ⍴ 1 2 5 6 4 3 , 0.5 ¯3 ¯4 ¯2
¯6 ¯6
1    2    5    6    6    6
0.5  0.5 ¯4   ¯2   ¯2   ¯2
```

---

**Title:** *sauce/pminreduce.1*

**Summary:** Partitioned ⌊/*array*; each 1 in the boolean left argument designates the beginning of a part in the array.

**Description:**
*result←boolean pminreduce array*

Partitioned ⌊/*array*. *boolean* is a vector with ⍴*boolean* ↔ ¯1↑⍴*array* (although scalar extension may be performed). Each 1 in *boolean* designates the beginning of a part of *array*.

Examples:

```
      1 0 0 0 1 0 1 0 pminreduce ¯1 2 ¯8 3 9 0.5 12 14
  ↔  (⌊/ ¯1 2 ¯8 3),(⌊/ 9 0.5),(⌊/ 12 14)


      1 0 1 0 0 0 pminreduce 2 6 ⍴ 1 2 6 4 5 3 , 0.5 ¯3 ¯5 ¯
4 ¯6 ¯6
 1  3
¯3 ¯6
```

---

**Title:** *sauce/pminscan.1*

**Summary:** Partitioned ⌊*array*; each 1 in the boolean left argument designates the beginning of a part in the array.

**Description:**
*result←boolean pminscan array*

Partitioned ⌊*array*. *boolean* is a vector with ⍴*boolean* ↔ ¯1↑⍴*array* (although scalar extension may be performed). Each 1 in *boolean* designates the beginning of a part of *array*.

Examples:

```
      1 0 0 0 1 0 1 0 pminscan ¯1 2 ¯8 3 9 0.5 12 14
  ↔  (⌊\ ¯1 2 ¯8 3),(⌊\ 9 0.5),(⌊\ 12 14)

      1 0 1 0 0 0 pminscan 2 6 ⍴ 1 2 6 4 5 3 , 0.5 ¯3 ¯5 ¯4
¯6 ¯6
1    1    6    4    4    3
0.5 ¯3   ¯5   ¯5   ¯6   ¯6
```

---

**Title:** *sauce/pnescan.1*

**Summary:** Partitioned ≠*array*; each 1 in the boolean left argument designates the beginning of a part in the array.

**Description:**
*result←boolean pnescan array*

Partitioned ≠*array*. *boolean* is a vector with ⍴*boolean* ↔ ¯1↑⍴*array* (although scalar extension may be performed). Each 1 in *boolean* designates the beginning of a part of *array*.

Examples:

```
      1 0 0 0 1 0 1 0 pnescan 1 1 0 1 1 0 0 1
  ↔  (≠\ 1 1 0 1),(≠\ 1 0),(≠\ 0 1)

      1 0 1 0 0 0 pnescan 2 6 ⍴ 1 0 1 0 1 0 , 1 1 0 1 1 1
1 1 1 1 0 0
1 0 0 1 0 1
```

It is assumed that ∧/,*array*∈ 0 1 ↔ 1.

---

---

**Title:** *sauce/porreduce.1*

**Summary:** Partitioned v/*array*; each 1 in the boolean left argument designates the beginning of a part in the array.

**Description:**
*result←boolean porreduce array*

Partitioned v/*array*. *boolean* is a vector with ⍴*boolean* ↔ ¯1↑⍴*array* (although scalar extension may be performed). Each 1 in *boolean* designates the beginning of a part of *array*.

Example:

```
      1 0 0 0 1 0 1 0 porreduce 0 1 0 0 1 1 0 0
  ↔  (v/ 0 1 0 0),(v/ 1 1),(v/ 0 0)

      1 0 1 0 0 0 porreduce 2 6 ⍴1,(10⍴0),1
1 0
0 1
```

It is assumed that ∧/,*array*∈ 0 1 ↔ 1.

---

**Title:** *sauce/porscan.1*

**Summary:** Partitioned v*array*; each 1 in the boolean left argument designates the beginning of a part in the array.

**Description:**
*result←boolean porscan array*

Partitioned v*array*. *boolean* is a vector with ⍴*boolean* ↔ ¯1↑⍴*array* (although scalar extension may be performed). Each 1 in *boolean* designates the beginning of a part of *array*.

Example:

```
      1 0 0 0 1 0 1 0 porscan 0 1 0 1 1 0 0 1
  ↔  (v\ 0 1 0 1),(v\ 1 0),(v\ 0 1)


      1 0 1 0 0 0 porscan 2 6 ⍴ 0 1 0 1 0 1 , 0 0 0 0 1 0
0 1 0 1 1 1
0 0 0 0 1 1
```

It is assumed that ∧/,*array*∈ 0 1 ↔ 1.

---

---

**Title:** *sauce/pplusreduce.1*

**Summary:** Partitioned *+/array*; each 1 in the boolean left argument designates the beginning of a part in the array.

**Description:**
*result←boolean pplusreduce array*

Partitioned *+/array*. *boolean* is a vector with *ρboolean* ↔ ¯1↑*ρarray* (although scalar extension may be performed). Each 1 in *boolean* designates the beginning of a part of *array*.

Examples:

```
      1 0 0 0 1 0 1 0 pplusreduce ¯1 2 8 3 9 0.5 12 14
  ↔  (+/ ¯1 2 8 3),(+/ 9 0.5),(+/ 12 14)

      1 0 1 0 0 0 pplusreduce 2 6 ρ 1 2 6 4 5 3 , 0.5 ¯3 ¯5
¯4 ¯6 ¯6
  3    18
 ¯2.5 ¯21
```

---

**Title:** *sauce/pplusscan.1*

**Summary:** Partitioned *+\array*; each 1 in the boolean left argument designates the beginning of a part in the array.

**Description:**
*result←boolean pplusscan array*

Partitioned *+\array*. *boolean* is a vector with *ρboolean* ↔ ¯1↑*ρarray* (although scalar extension may be performed). Each 1 in *boolean* designates the beginning of a part of *array*.

Examples:

```
      1 0 0 0 1 0 1 0 pplusscan ¯1 2 8 3 9 0.5 12 14
  ↔  (+\ ¯1 2 8 3),(+\ 9 0.5),(+\ 12 14)

      1 0 1 0 0 0 pplusscan 2 6 ρ 1 2 6 4 5 3 , 0.5 ¯3 ¯5 ¯4
 6 6
1    3    6   10   15   18
0.5 ¯2.5 ¯5  ¯9  ¯3    3
```

---

**Title:** *sauce/preverse.1*

**Summary:** Partitioned ϕ*array*; each 1 in the boolean left argument designates the beginning of a part in the array.

**Description:**
*result←boolean preverse array*

Partitioned ϕ*array*. *boolean* is a vector with ⍴*boolean* ↔ ¯1↑⍴*array* (although scalar extension may be performed). Each 1 in *boolean* designates the beginning of a part of *array*.

Examples:

```
      1 0 0 0 1 0 1 0 preverse 'abcdxymn'
  ↔  (ϕ'abcd'),(ϕ'xy'),(ϕ'mn')

      ⎕io←1
      1 0 1 0 0 0 preverse 2 6 ⍴⍳12
2  1  6  5  4  3
8  7 12 11 10  9
```

*array* may contain characters, numbers, or enclosures.

---

**Title:** *sauce/proindexof.1*

**Summary:** Progressive dyadic iota; e.g. *'aba' proindexof 'acaaba'* ↔ ⎕*io*+ 0 3 2 3 1 3.

**Description:**
*indices←sequence proindexof vector*

Progressive dyadic iota.
E.g. *'aba' proindexof 'acaaba'* ↔ ⎕*io*+ 0 3 2 3 1 3.

The arguments may contain characters, numbers, or enclosures.

The result varies appropriately according to the value of ⎕*io*.

---

**Title:** *sauce/recurrence*.1

**Summary:** *result*[*i*]←*leftarg*[*i*]+*rightarg*[*i*]×*result*[*i*-1] ⋄ *result*[⎕*io*]←*leftarg*[⎕*io*].

**Description:**
*sequence*←*addend recurrence multiplier*

*sequence*[*i*]←*addend*[*i*]+*multiplier*[*i*]×*sequence*[*i*-1] ⋄ *sequence*[⎕*io*]←*addend*[⎕*io*].

It is assumed that ~0∈*multiplier*.

---

**Title:** *sauce/removetrail*.1

**Summary:** Removes trailing columns or elements from an array.

**Description:**
*compressed*←[*elements*] *removetrail array*

Removes all trailing columns consisting solely of *elements* from *array*.

Default *elements* is 1↑0⍴*array* (i.e. 0 or '').

Examples:

```
      removetrail '  ab c  d   ' ↔ '  ab c  d'

      '∘⎕' removetrail 'abc∘∘∘   ↔  'abc
                       ⎕∘x∘∘⎕       ⎕∘x
                       ⎕⎕⎕⎕⎕⎕'      ⎕⎕⎕'
```

---

**Title:** *sauce/rightjustify.1*

**Summary:** Right justifies each row of an array.

**Description:**
*result*←[*element*] *rightjustify array*

The default *element* is 1↑0⍴*array* (i.e. 0 or ''). The function right justifies each row of *array* with respect to *element*. Note: it will work even if *element* has more that one distinct element; in this case, right justification is done with respect to all the elements of *element*.

```
E.g.  rightjustify 'joe      ↔   '  joe
                    mary           mary
                   john           john
                   ed   '           ed'

E.g.  ' ∘' rightjustify 'joe∘∘∘   ↔  '∘∘∘joe
                         ∘∘mary        ∘∘mary
                         ∘john          ∘john
                         ∘ed  ∘'         ∘∘ed'
```

*array* may contain characters, numbers, or enclosures.

---

**Title:** *sauce/runlengths.1*

**Summary:** E.g. *runlengths* 1 9 9 4 4 4 9 ↔ 1 2 3 1.

**Description:**
*lengths*←*runlengths vector*

E.g. *runlengths* 1 9 9 4 4 4 9 ↔ 1 2 3 1.

*vector* may be character or numeric.

---

---

**Title:** *sauce/squeeze.1*

**Summary:** Removes all leading, trailing, and redundant elements (spec'd in the left argument) from a vector.

**Description:**
*compressed←[element] squeeze vector*

Default *element* is 1↑0⍴*vector* (i.e. 0 or ''). The result is like *vector*, but with all leading, trailing, and redundant *element*s removed.

Example:

```
      squeeze '  ab      c  de f ' ↔ 'ab c de f'
```

*vector* may contain characters or numbers; if *element* is specified, *vector* may also contain enclosures.

---

**Title:** *sauce/stringfind.2*

**Summary:** Finds all occurrences of one vector in another.

**Description:**
*indices←vector stringfind substring*

Returns the index of the beginning of each occurrence of *substring* in *vector*.

Examples:

```
      ⎕io←1
      'banana' stringfind 'ana'
2 4
      ⍴⎕←'banana' stringfind 'nan'
3
1
      'banana' stringfind ''
1 2 3 4 5 6 7
```

[*vector stringfind* '' ↔ ⍳1+⍴*vector* as suggested by A.D. Falkoff in "A Note on Matching: Where Do You Find the Match to an Empty Array?", APL79 Conference Proceedings.]

```
      ⎕io←0
      'banana' stringfind 'ana'
1 3
```

*string* may be character or numeric.

The result varies appropriately according to the value of ⎕*io*.

---

**Title:** *sauce/vtom.1*

**Summary:** Returns a matrix with one string per row, where each string in the right argument is preceded by 1↑*rightarg*.

**Description:**
*matrix*←[*element*] *vtom vector*

(Vector TO Matrix.) The function returns a left-justified matrix with one string per row, where each string in *vector* is preceded by 1↑*vector*. *element* specifies the fill element used for justification; the default *element* is 1↑0⍴*vector* (i.e. 0 or ' '). *vector* may be contain characters, numbers, or enclosures.

Examples:

```
      vtom ' the kitten ran away.'
the
kitten
ran
away.

      '*' vtom '∘hey∘diddle∘∘diddle∘∘∘'

hey***
diddle
******
diddle
******
******
******
```

*sauce/vtom* is frequently used with *sauce/squeeze*, as follows.

```
      vtom ' ',⎕←squeeze ' john   george ed     ted   '
john george ed ted
john
george
ed
ted
```

---

**Title:** *wgr/ftt.2*

**Summary:** Forms formatted representation of timestamps from ⎕*rdci* form. Allows vector input.

**Description:**
Similar to function *ftt* in *ws* 1 *ts* but allows vector input and returns matrix result. Argument is vector of timestamps as encoded numeric vectors from ⎕*rdci*[3] result is array of timestamps as numeric matrix.

---

**Title:** *wgr/packshow*.1

**Summary:** Function to display contents of package (including functions and other packages) in neat form.

**Description:**
*packshow* is a single function that displays as a heirarchial structure, the contents of a package given as argument. The right argument is a package. The optional left argument is a character string used to name the root level of package. If no left argument is given and right argument is package, no root name is used. Functions are extracted and displayed in 1 ⎕*fd* form ,variables are preceded by shape vector in assignment form. Undefined names are represented by *?*. Locked functions by ⍢

---

**Title:** *wgr/ttf*.1

**Summary:** Forms ⎕*rdci*[3]-format-encoded timestamp vector from matrix of ⎕*ts*-form timestamps.

**Description:**
Similar to function *ttf* in ws 1 *ts* (on SHARP APL/370 systems) but allows matrix input and returns vector result. Argument is vector of timestamps as numeric arrays from ⎕*ts*. Result is vector of timestamps as encoded numeric vector.

---

**Title:** *yudi/fill*.1

**Summary:** Fills a character string with a delimiter for a specified number of times.

**Description:**
*result*←<*del*Δ*no*Δ*pos* fill *string*

*fill* is a character string filler. It can be used to fill a character vector at positions *pos* with a delimiter *del* for an integer *no* of times. The left argument is a 3 element enclosed array containing *del*, *no*, and *pos*. The right argument is a character vector. The result is *no* characters longer than the original string. This function is ⎕*io* independent and assumes anything out of range is either tagged on to the beginning or the end of the character *string*.

E.g. `(' '⊃3⊃(string=' ')/⍳⍴string) fill string←'a string of characters to fill'`
*a string of characters to fill*

---

## Appendix G: INSTALLING THE APL CHARACTER ROM

This appendix describes how to install the I.P. Sharp APL character ROM in an IBM Personal Computer. This ROM is the read-only memory that defines APL characters for the video display. It replaces the character ROM that IBM supplies with the Personal Computer. (For guidance on whether to install the ROM, see chapter 2 **Getting started**.)

This procedure applies to the monochrome display/parallel printer adapter or the color/graphics adapter in the IBM Personal Computer, Personal Computer XT, Personal Computer XT/370, or Personal Computer AT.

**EVERY STEP IN THIS PROCEDURE IS IMPORTANT**
**READ THE PROCEDURE CAREFULLY BEFORE YOU BEGIN**

1 Assemble your tools. Two screwdrivers should be enough: a small one to be used as a lever; and a larger one (with a quarter-inch blade) for the computer's fasteners. A dish or bowl (or hubcap) can temporarily store the fasteners you're going to remove.

2 Remove any diskettes, turn off the system unit and monitor, and unplug them.

3 See where each cable is plugged into the system unit, and make notes or pictures to help you remember--**you will have to put it back together**--then remove all cables from it. (You can use the back page for your notes or pictures.)

4 If the video display is sitting on the system unit, set it out of the way.

5 Turn the system unit so that its back faces you.

6 Remove the **large** hexagonal screws at the back of the system unit. There are two, three, or five such screws, depending on the model.

7 Slide the system unit's case away from you, tilt it up, and remove it.

8 Clear a flat, clean surface for the display adapter board. You may want to cover the surface with scrap paper to protect the finish.

9 Identify the adapter you want to remove: this is the expansion board mounted vertically front to back that the video display was plugged into; it is not the mother board, which is horizontally mounted at the bottom of the system unit.

10 Remove the adapter board. Unscrew and remove its holddown clamp, and carefully jiggle the board out of the bus (the gold comb-like connector with the mother board). A slight outward pressure on the back of the system unit may help. Don't pull

one end of the board too much farther out of the unit at a time than the other.

11 Lay the board on your working surface, components upward, bracket to the right.

**Static electricity:** A static discharge from your fingers could destroy the character ROM or other components on the adapter. If you think you're carrying a charge, get rid of it, for example, by touching a waterpipe or a tap.

Never ground the character ROM or the adapter (by, for example, touching them to a metal table) while you're holding them.

12 Remove the I.P. Sharp APL character ROM from its box, and lay it on the work surface.

13 Identify the character ROM.

On the monochrome adapter, the character ROM is in the centre of the board and stands twice as high as any other component.

On the color adapter, the character ROM is the 5th chip from the left, spanning the second and third rows of four.

On either adapter the character ROM is the only chip in the socket (marked U33), and the only chip the same size as the APL character ROM. The IBM character ROM may be marked something like:

578805
(c) IBM 1981
(M)XE8201

14 You're now ready to remove the IBM character ROM. Note the indentation on the socket: this is how the APL ROM must be oriented. Slide the small screwdriver between the ROM and the socket, at one end of the ROM. Gently lever the ROM up a millimetre. **The action may be stiff.** Slide the screwdriver between the ROM and the socket at the other end; gently lever that end up. Repeat this until the ROM is free. Never let one end get too much further out than the other.

15 Put the removed ROM in the box that the I.P. Sharp APL character ROM came in. (You may want it later, so don't throw it out.)

16 Some of the socket springs may have popped out; push them back in. (A socket spring is the bent bit of metal that holds one of the ROM's pins in the socket.)

17 With one hand, pick up the I.P. Sharp APL character ROM; with the other, touch the comb-like bus on the adapter board.

This touching equalizes the electric potential of the board and the ROM, and prevents damage to the ROM.

18 As the ROM is new, its pins may be splayed. If the ROM doesn't fit, lay the character ROM on the working surface, resting on one row of pins. Gently roll the ROM until the pins are perpendicular to the chip. Repeat for the other row of pins.

19 You're now ready to insert the I.P. Sharp ROM in the now empty socket. This is the most delicate part of the procedure.

**Orient the ROM so that the indentation on the chip matches the groove on the socket and the orientation of most other components on the board.**

20 Insert the tips of all the pins on one side of the ROM into the available holes.

21 Insert the pins on the other side into the corresponding holes. Use your fingers.

22 Jiggle the ROM until the pointed ends of the pins can't be seen. **Make sure all the pins are inserted.**

23 Firmly but gently push the ROM into place. Check that no pin is bent under the ROM. (If one is, remove the chip [as in step 14]; unbend the pin; and resume at step 19.)

24 Put the display adapter board back into its expansion slot. Be careful to insert the board into the guide at the front of the system unit.

25 Screw down the holddown clamp.

26 Put the cover back on the system unit, and bolt it down.

27 Finish up: turn the system unit around, put the monitor back in place, plug in all cables; the power cable last. Check that the monitor cable is plugged in right side up. For information about cable hookup, see the **Guide to Operations** [IBM publication 6025003].

28 You can now display the SHARP APL character set on your PC.

A

abort, ACOM command return code, 12-17
Abort, Retry, Ignore?, message displayed by DOS, 11-2
access control, APL file, 4-9
  __, shared variable, 9-10
  __, shared variable, from user-written AP, 10-11
access control, shared-variable, 9-10, 10-11
  __, meaning of possible settings, 9-17
  __, meaning of the four elements, 9-12
  __, rejection of attempted reference to shared variable by user-written AP, 10-13
access matrix, functions to read and set, 4-10, 8-5, 8-7
account number, set as start parameter, 2-15
  __, user who wrote file component, 8-6
accounting, text in APL, E-3
ACOM, auxiliary processor, 1-3, 5-1, 12-1
  __, buffer utilization and transmit/receive commands, 12-9
  __, PC workspace, 5-1
  __, start parameters, 2-13, 2-14
  __, summary of commands, 12-2
  __, workspace on mainframe system, 5-1
ACOM.EXE, asynchronous communications AP, 2-9
*acom*, timeshared system workspace to communicate with PC using ACOM, 12-1, 12-7
action, control keys, A-6
active workspace, size, 7-2
adjacent field, restrictions in AP124 for 3270 terminals, 13-17
alarm, FSCR command to sound, 13-28
  __, sounded during full-screen display, 13-3
*alert*, defined function in WS *fscr*, 13-35
algebra, text in APL, E-3
*alloceq*, utility function for equitable allocation within limits, F-30
*allocfifo*, utility function to allocate by FIFO, subject to limits, F-31
alphabets, representation of, C-2
already signed on, SVP return code to user-written AP, 10-14
<Alt>, 3-1, 6-4, A-2
  __, summary of actions, A-6
<Alt Backspace>, overstrike, 3-2
<Alt Del>, delete balance of line, 6-8
  __, □*arbin*/□*arbout* code, 6-22
  __, within definition of function key, 6-21
<Alt Down>, 3-12
<Alt End>, 3-12
<Alt Home>, 3-12
<Alt Keypad 5>, to fix reference line in log, 3-12, 6-15
<Alt NumLock>, enable/disable window wrap, 3-14, 6-15
<Alt ScrollLock>, request output curb, 3-15, 6-15
alternate case, 3-1, 3-14, 6-4, A-2
alternation between entry and display, full screen, 13-2
analysis, text in APL, E-2
Anscombe, F.J., E-3
any name, symbols for, in DOS file request, 11-5
APDEFN.H, interface to SVP in C language for user-written auxiliary processors, 10-1

APDEFN.PCC, interface to SVP in assembler language for user-written APs, 10-1
APDEFNA, source file of definitions governing user-written APs, 10-1, 10-2, 10-6, 10-8, 10-13
APL, characters in names shared with user-written AP, 10-6
__, implementation on IBM PC, 1-1
__, keyboard location of special characters, 1-5, 6-5, A-2
__, priority under task dispatcher, 10-2
__, starting the system, 2-10, 2-11
__, translation to and from ASCII in DOSX file operations, 11-5
APL.BAT, batch file for starting APL, 2-8
APL character generator, diagram of glyphs, B-6
APL files, characteristics, 8-1
APL keyboard, relation to union keyboard, 6-6
APL Language, IBM publication GC26-3847, E-1
APL Quote Quad, E-4
APL Users Meeting, E-4
append, APL object as new component of file, 4-3, 8-4
AP124, compatibility with FSCR, 13-1
argument error, SVP return code to user-written AP, 10-14
arguments, to machine-language program called under DOSX, 11-18
arrow, in status line, to indicate which window current, 8-4
ASCII, characters in names shared with user-written AP, 10-6
__, translation to and from APL in DOSX file operations, 11-5
ASCII keyboard, C-2, C-3
ASDATA, format of APL data shared with APL via SVP, 10-6
__, type of structure (control block) in user-written auxiliary processor, 10-4, 10-6
aspect ratio, height and width of character position, 13-16
assembler, preparing program for execution by DOSX *call*, 11-18
assembler language, program, 11-16
__, user-written APs, 10-1
assignment, setting shared variable, 9-4
asterisk, display in status line when more than 10 statements in typeahead buffer, 6-9, 6-13
__, to mean "any name" in DOS file request, 11-5, 11-8
asynchronous communication, auxiliary processor, 1-3, 12-1
attention, event 1002 signalled by <Ctrl ScrollLock>, 6-16, 6-18
ATTN, ACOM command to transmit break, 12-11
attributes, FSCR commands to reset attributes of individual characters, 13-29
__, FSCR commands to reset attributes of fields, 13-27
autoanswer timeout, tolerable delay to start communication with ACOM, 12-3
autoload, workspace at startup , 2-16
autoskip, parameter in FSCR field description, 13-21, 13-25, 13-27
auxiliary processor, concept, 9-2
__, start when APL started, 9-2
__, user-written, 10-1
auxiliary processors, 1-3, 9-1, 10-1
availability, APL file system, 4-9, 8-4
AX, address for return code of call to SVP, 10-7

B

B, FSCR start parameter: buffer size, 2-14

__, SFS start parameter: number of diskette buffers, 2-15
B-task, difference between PC and 370, D-3
B (CP8X) start parameter, 2-13
B (File System start parameter), 2-15, 8-2
B (FSCR start parameter), 2-14, 13-40
B (SVP start parameter), 2-16
backquote, keyname, A-2
backslash, keyname, A-2
backspace, deletion, 3-4
__, destructive, 6-8
__, to form overstrike, 6-7
<Backspace>, 13-7, A-2
__, summary of actions, A-6
Backspace key, 3-1
backup files, user responsibility, D-2
barchart, 3-9
Bartoli, G., E-5
base case, 6-4, A-2
BASIC, compared with APL in sample program, E-5
batch file, for starting APL, 2-8
__, sequence in which APL components started, 2-13
baud rate, in ACOM start command, 12-3
beeper, PC "alarm", FSCR command to sound, 13-28
BELL, to mark end of transmission, 12-7
bell, FSCR command to sound, 13-28
__, sounded during full-screen display, 13-3
Berry, P., E-5
bibliography, E-1
BIN, file required by DOSX *call*, 11-18
BIOS functions, disallowed from interrupt task of user-written AP, 10-2
bits per character, in ACOM start command, 12-4
blank, <Keypad +> while using numeric keypad, 6-8
blank at end of field row, contrast between PC and IBM 3270 treatment, 13-19
blank keyboard, diagram, B-9
blank processing, parameter in field description, 13-21
blank vs. null, row-trailing positions in full-screen field, 13-18
blanks, trailing, 13-21, 13-25, 13-27
blink, value of FSCR "highlighting" attribute, 13-25
blinking cursor, to form overstrike, 6-7
blocked paths, shared-variable transition diagram, 9-15
blue, FSCR colour, 13-24
box-drawing characters, D-5
brackets, angle, indicating keys, 0-12
break, ACOM return code, 12-17
__, signal transmitted by ACOM command, 12-11
<Break>, events 1002 and 1003 signalled by pressing <Ctrl ScrollLock>, 3-3, 6-16, 6-18
__, shared variable, interrupt signalled by pressing <Ctrl ScrollLock>, 9-16
Bryson, S., E-4
*bstranslate*, utility function to resolve vector containing backspaces to composite characters, F-9
buffer, FSCR, storage of data for display, 13-6
__, FSCR, unaffected by change in field format, 13-26

buffer overflow, ACOM return code, 12-17, 12-18
buffer size, startup parameter for file system, DOSX, SVP, FSCR, 2-15
buffered input, full-screen mode, 13-10
buffered write, FSCR command, 13-28
building APL system, procedure, 2-8
bytes, required to store a file, 8-10
  __, size of a DOS file, 11-8

C

C, display in status line while execution curbed, 6-9, 6-13
  __, programming language, in user-written APs, 10-1
  __, SUPER start parameter: argument to CHDIR to select directory, 2-16
C (CP8X start parameter), 2-13
calculus, text in APL, E-3
*calendar*, utility function, F-2
*call*, DOSX command to execute machine-language program, 11-16, 11-18
CALLTEST, DOS utility program to test machine-language program, 11-18, 11-19
capital letters in names, 3-2
CAPS, keyboard entry state, 3-4, 6-7, 6-13
caps reversal, 6-7, 6-15
<CapsLock>, 3-2, 13-7
  __, summary of actions, A-6
caret, to mark insertion state, 3-4, 6-13
carriage return vs. newline, in translations between ASCII and APL, 11-6
case, lower, and caps in names, 3-6
  __, lower, shift, alt, control, 6-4, A-2
*catenaterows*, utility function to join vertically, padding as necessary, F-31
*centre*, utility function to center text on a line , F-23
change directory, DOSC function call, 11-14
character data, 3-8
character generator, APL, 1-5
  __, APL, diagram of glyphs, B-6
  __, IBM, diagram of glyphs, B-7
character input, symbol for, 3-8
character set, session manager's tables, B-1
  __, union, C-1
  __, with APL ROM installed, 2-6, A-11
  __, with IBM rom installed, A-11
characters, maximum number for ACOM receive command, 12-7
CHDIR, DOS command, 2-11
*chdir*, DOSX command to change directory, 11-14
checksum, ACOM data transmission, 12-6
  __, algorithm used by ACOM, 12-9
checksum error, ACOM return code, 12-18
*ch∆ravel*, utility function to ravel array, with embedded CRs , F-8
circuit theory, text in APL, E-3
)*clear*, 3-11
  __, effect on shared variable, 9-6

<Clear>, correspondence between PC keys and 3270 keys, 13-10
__, possible result of FSCR command 3, 13-30
clear input stage, □*arbin* code, 6-22
clear to send, reported by ACOM, 12-20
clock, user responsibility, D-2
clone ID, auxiliary processor, 10-4
__, effect on □*sc*, 9-18
__, in order to make shared variable offer, 9-6, 9-7
close, DOS file, 3-11, 11-4, 11-8
*closefx*, utility function to close printer pseudofile, F-14
codes, return, DOSX, 11-23
__, return, FSCR, 13-37
*col*, function to arrange data in column, 3-8
*color*, defined function in WS *fscr*, 13-34
colour, attribute of FSCR field, 13-24
__, FSCR command to re-set, 13-27
colour/graphics adapter, with IBM PC, 1-5
column width, formatted numeric data, 3-11
*columnize*, utility function to arrange page in multiple columns, F-24
columns, invisible until window moved left or right, 6-12
comma, keyname, A-2
command, syntax for DOSX, 11-1
__, transmitted by shared variable, 13-13
command summary, FSCR, 13-36
commands, DOSX, classification, 11-4
__, DOSX, summary table, 11-22
communication, asynchronous ports, 5-1
__, role of shared variables, 9-1, 9-11
communication with auxiliary processor, shared variable, 13-5
compatibility, SHARP APL on PC and on 370, 1-3
complex numbers, 1-2
component, file, 4-1, 4-4
__, file: saver, component size, timestamp encoded in control information, 4-9
component number, effect of dropping preceding components, 4-7
composite character, formed by overstriking, 6-7
composite characters, overstriking to form, 3-2
*compsΔreprt*, utility function to report on specified file component, F-15
concurrent processes, task dispatcher, 10-2
concurrent tasks, shared use of files, 4-9
consecutive integers, 3-7
consulting, I.P. Sharp Associates, 1-5
)*continue*, effect of shared variable, 9-19
__, system command, effect on tied files, 4-5
control action, by key, A-6
control block, auxiliary processor, 10-4
control case, 6-4
control characters, translation when sent to parallel printer, 11-11
control information, file component, 4-9, 8-6
control keyboard, diagram, A-5
control keypad, 6-6
conventions, 0-12
COPIER.EXE (program), 2-5
COPY, DOS program, 2-9, 2-11

copy, line from log to stage, 3-5, 6-10
__, multiple lines from log to stage, 3-9
)*copy*, difference between PC and 370, D-6
copy data transfer, SVP command, 10-11
copy intial selection, SVP command, 10-11
copy shared variable, intermediate step in certain SVP data transfer operations, 10-12
copying diskettes, with one drive, 2-3
__, with two drives, 2-2
countries, sample data, 3-8, 3-9
*cpu*, utility function to measure cpu usage of expression, F-3
*cpu*, utility function to measure execution, F-2
cpu stop, <Ctrl NumLock>, 6-16
)*cpulim*, difference between PC and 370, D-6
CP8X.EXE (program), 2-8
__, start parameters, 2-13
crash recovery, difference between PC and 370, D-1
create, DOS command to create new file, 11-7
__, native file, 3-10, 11-3
__, new APL file, 4-3, 8-4
CTOH.ASM, sample assembler file of machine-language program invoked under DOSX, 11-19, 11-20
CTOH.BIN, sample binary file of machine-language program invoked under DOSX , 11-19
<Ctrl>, 6-4, A-2
__, summary of actions, A-6
<Ctrl Break>, 3-15
__, interrupt computation or display, 3-3
__, strong interrupt signal, 3-10
<Ctrl CapsLock>, to reverse keyboard caps, 6-7
<Ctrl Del>, delete typeahead buffer, 6-9
<Ctrl NumLock>, keyboard signal to halt all processing, 6-18
CTS, reported by ACOM, 12-20
CTS/RTS, flow control, 12-4
cumulative product, 3-7
cumulative sums, 3-7
CURB, status line indicator that program curb has been requested, 6-13, 6-18
curb, request to permit no new output by <Alt ScrollLock>, 6-15, 6-18
current count, index and translation, DOS file command, 11-8
current window, selection of upper or lower, 6-11, 6-15
cursor, FSCR command to set position, 13-28
__, mark position on screen at which next keyed character will appear, 3-3
__, move to beginning or end of stage, 3-12
__, skip to next field, 13-21
cursor control keys, 3-4
cursor location, reported during full-screen entry, 13-3
cursor movement, control during full-screen entry, 13-9
__, control to edit input stage, 6-8
cursor position, reported in result of FSCR command 3, 13-30
__, to indicate choice, 13-1
customize, BAT file to specific APL configuration, 2-13
cut text into words, 3-8

*cyclicmesh*, utility function to group nth elements of an array, F-32

D

D, DOS function, 11-13
Dalois, M., E-2
damage, erasing DOS file while it is open, 11-9
  __, file directory, from removing diskette while file tied, 4-4, 8-1
  __, produced by incorrect auxiliary processor, 10-1
data, larger than full-screen field, 13-17
  __, transmitted by shared variable, 13-12, 13-14
data bits per character, in ACOM start command, 12-4
data set ready, reported by ACOM, 12-20
  __, response to ACOM start command, 12-4
data terminal ready, effect of ACOM stop command, 12-13
data transfer, SVP operations available to user-written AP, 10-11
data validation, during transfer between APL and user-written AP, 10-6
data variable, two-variable interface to FSCR, 13-15
*datatype*, utility function to report internal type, F-3
date, encoded in control information of APL file component, 4-9
*datedecode*, utility function to return standard dates from Julian daynumbers, F-9
*dateencode*, utility function to calculate Julian daynumbers from standard dates, F-9
*daterep*, utility function to return 3-element representation of julian day numbers, F-13
*day*, utility function to calculate day of week from date, F-24
DEBUG, DOS, 11-16, 11-18
decimal point, <Ins> key while using numeric keypad, 6-8
decimals, formatted numeric data, 3-11
decode error messages, workspace *fscrex*, 13-38
default, clone ID, 9-7
  __, count and translation for DOS file read/write, 11-6
  __, keyboard state, return to with <Esc>, 6-14
default drive, DOSX function to report or select, 11-13
default state, screen and keyboard under session manager, 6-14
define function keys, function *setfkeys* in workspace *fkeys*, 6-20
  __, ⎕*arbin* procedure, 6-20
definition, user-defined function, 3-7
degree of coupling, inquiry by user-written AP, 10-10
  __, shared variable, 9-7
<Del>, during editing of input stage, 3-4, 6-8
  __, during full-screen entry, 13-9
  __, summary of actions, A-10
delay, attempt to access interlocked shared variable, 9-16, 9-17
  __, avoided by functions ⎕*ws*, ⎕*pack*, etc., 9-16, 9-17
  __, avoided by inspection of ⎕*svs*, 9-16, 9-17
  __, interrupting, 9-16, 9-17
deletion, balance of line, 3-6
  __, from session log, 6-1
  __, within row of a full-screen field, 13-18
delimiter, in argument to *setfkeys*, 6-20, 6-22
Dell'Aquila, C., E-5
density of population, example of computation, 3-9

destructive backspace, 6-8
diagram, blank keyboard, B-9
__, control keyboard, A-5
__, keynumbers, B-8
__, main keyboard, A-4
diagram summarizing shared variable communication, 9-14
dialogue, full screen, 13-2
differences, PC and 3270 versions of full-screen AP, 13-33
__, PC and 370 environments for APL, D-1
DIR, difference between DOS and DOSX commands to read directory, 11-8
*dir*, DOSX file command, 11-8
directory, DOS, display of, 3-9, 11-8
__, make, change, or remove, DOSX function calls, 11-14
discard file, see "erase", 4-6
disk drive, needed to run SHARP APL/PC, 1-5
disk drives, number of, 11-13
disk operating system, DOS, 3-9
DISKCOMP, DOS program to verify copying, 2-3
DISKCOPY, DOS program, 2-2, 2-4
diskette, copying, 2-2
__, directory damage by removing while file tied, 4-4, 8-1
__, list of files in SHAP APL/PC reference diskette, 2-7
dispatcher, concurrent tasks, 10-2
display, monochrome vs. colour/graphics, 1-5
__, value of a variable, 3-6
*distinct*, utility function to remove duplicates from an array, F-32
*distinctrows*, utility function to remove duplicate rows from matrix, F-33
distribution diskettes, copying, 2-2
*distround*, utility function to round elements preserving a fixed sum, F-33
division table, 3-7
*dno*, utility function to return Julian day numbers from array of dates, F-13
*do*, function to execute single FSCR command via two-variable interface, 13-15
*dom*, function to pass multiple commands to FSCR, 13-15
*domain error*, 3-3
DOS, file identifier, 4-2
__, operating system, communication with, 3-9
__, user's guide, E-5
__, version required to run SHARP APL/PC, 1-5
__, workspace identifiers, 7-1
DOS filenames, APL files, 8-2
DOS function calls, from DOSX, 11-13
DOS functions, disallowed from interrupt task of user-written AP, 10-2
DOS Interface Auxiliary Processor, DOSX, 11-1
DOS TERMINATE, prohibition on use by AP, 10-16
DOSX, auxiliary processor, 1-3, 3-9, 11-1
__, start parameters, 2-14, 11-2
__, summary of commands, 11-22
__, table of return codes, 11-23
DOSX.EXE, DOS interface AP, 2-9

DOSX commands, classification, 11-4
<Down>, cursor movement, 3-5, 6-4, 6-9
  __, keyname, A-3
  __, movement on window, 6-11
  __, summary of actions, A-9
download files, mainframe to PC, 5-1
drive, as part of name of a file, 4-8, 8-6
  __, default, DOSX function to set or report, 11-13
  __, optional parameter in DOS file command, 11-5
drive number, to identify file, 4-2, 8-5
  __, to identify workspace, 7-1
*drives*, DOSX function to report number of disk drives, 11-13
drop, remove components from end of file, 4-7, 8-4, 8-10
DSR, reported by ACOM, 12-20
  __, response to ACOM start command, 12-4
DTR, turned off by ACOM stop command, 12-13
<DUP>, correspondence between PC keys and 3270 keys, 13-10
duration, argument to MUSIC AP, 10-17
  __, full screen display, 13-3
Dutch language, APL publication, E-2
dynamic window, 3-12, 6-3

E

E, DOS function, 11-13
*each*, utility function to execute expression on each of a vector of enclosures, F-3
*easytie*, utility function to tie a file with provision for subsequent untie, F-25
*easyuntie*, companion utility function to *easytie*, F-25
edit, full screen display, FSCR command, 13-29
editing, full-screen display , 13-3
  __, incorrect entry on the stage, 3-3, 6-8
  __, produced by definition of function key, 6-21
editing keypad, 3-3
effective access control, how computed from control proposed by partners, 9-11
eight-bit data, transparent translation in ACOM transmit/receive command, 12-6
*elemreplace*, utility function to replace all occurrences of a scalar by a vector, F-33
*Emsgs*, variable, WS *fscrex*, 13-39
emulator, in implementation of SHARP APL/PC, 1-1
enclosed array, to permit single-variable interface to AP, 13-11
enclosed arrays, 1-2
*encvecfrmat*, utility function to produce enclosed vectors from matrix, F-15
<End>, □*arbin*/□*arbout* code, 6-22
  __, summary of actions, A-9
  __, to move cursor within stage, or window over log, 3-12
  __, within definition of function key, 6-21
end of transmission, character to mark, 12-7
*endspose*, utility function to move specied axes to end, F-4
*Enos*, variable, WS *fscrex*, 13-38
entry of new instruction, while earlier ones still in execution, 3-7
Epson printer, 11-11

equals, keyname, A-2
erase, APL file, 4-6, 8-4
  __, DOS file, 3-10, 11-4, 11-9
)*erase*, objects in workspace, 3-11
error messages, FSCR, function to decode, 13-38
*errormsg*, defined function, WS *fscrex*, 13-38
<Esc>, reset session-manager state, 3-3, 3-6, 3-12, 6-14
  __, summary of actions, A-6
Escape, to restore state of keyboard, 3-3, 3-6, 3-12, 6-14
escape sequence, translation when sent to parallel printer, 11-11
evaluated input, effect on full screen display, 13-3
even parity, in ACOM start command, 12-4
events, produced by errors in use of APL files, 8-8
exceptional routes, shared-variable transition diagram, 9-15
EXE, file produced by assembler, 11-18
EXE2BIN, DOS program used to prepare executable BIN file from assembler, 11-18, 11-19
exponential form, 3-6
*extendparts*, utility function to generate expansion masks, F-34
extension, optional portion of DOS filename, 11-5
  __, to identify SHARP APL workspace names to DOS, 7-1

F

F (FSCR start parameter), 2-15, 13-40
Falkoff, A.D., E-2
*fappend*, utility function to append to file, with resize as needed, F-16
*fappendr*, cover function for □*appendr*, F-16
*fcompare*, utility function to compare two files, F-16
*fcompress*, utility function to compress components from a file, F-17
*fcopy*, utility function to copy part or all of file, F-18
*fdelete*, utility function to delete file components, F-4
*fdrop*, utility function to drop file components, F-18
*ferase*, utility function to erase a file, F-18
*fexpand*, utility function to expand a file, F-19
*fextent*, utility function to report size and shape of file components , F-19
field, area of screen for input or display via FSCR, 13-6, 13-16
  __, full screen, minimum separation on 3270 terminals, 13-1
  __, modified, reported by FSCR command 3, 13-30
field description, FSCR format matrix, 13-24
field formats, FSCR command to set, 13-26
<Field mark>, correspondence between PC keys and 3270 keys, 13-10
field number, indentification of fields in FSCR, 13-16
field row, skip at end, 13-21
  __, term used in FSCR, 13-18
field width, formatted numeric data, 3-11
*fieldattr*, defined function in WS *fscr*, 13-34
*fieldtype*, defined function in WS *fscr*, 13-34
file, access control, 4-10
  __, APL, 4-1
  __, closed at shutdown of user-written AP, 10-16
  __, compared with workspace for storage, 4-1

__, damage by system reset while open, 6-16
__, diskette for storage, 2-8
__, erase, 4-6
__, identifier, 4-2, 8-1
__, list of APL files on diskette, 4-5
__, name of, 4-2
__, native, created under DOSX, 11-3
__, physical storage, 8-10
__, read components, 4-5
__, reservation, 4-4
__, tie, 4-3
*file access error*, 4-10, 8-7
file backup, user responsibility, D-2
*file component damaged*, 8-8
*file damaged*, 8-8
*file full*, 8-8
file functions, summary table, 8-12
file hold, difference between PC and 370, D-3
file identifier, rename, 8-6
*file index error*, 8-8
*file name error*, 8-8
*file quota used up*, 8-8
File System, start parameters, 2-15, 8-2
file system, procedure to start, 4-9
__, whether running, 4-9
*file system error*, 8-8
*file system no space*, 8-8
*file system not available*, 8-8
*file tie error*, 8-8
*file tie quota used up*, effect of SFS start parameter, 8-9
*file tied*, 8-8
Files, 1-2, 4-1, 8-1
files, number that can be simultaneously tied, 8-9
*filespace*, reports names and sizes of all files in specified library, F-26
fill, unused positions in full-screen field, 13-17
*fill*, utility function to fill a character string with a delimiter, F-55
*fill*, utility function to make specified insertions of a fill-sequence into a character vector, F-57
*findex*, utility function to select from file by array of indexes, F-20
Finnish APL Society, E-5
*finsert*, utility function to insert new componets in file , F-20
first character timeout, in ACOM transmit/receive command, 12-7
*firstones*, utility function to mark first of each group of 1s in a vector, F-34
*firstzeroes*, utility function to keep only the first of each group of 0s in a vector , F-34
*flib*, utility function to report alphabetized file library, F-20
flow control, in ACOM start command, 12-4
*fmatch*, utility function to match each component of a file with every of another, F-21
*fmtrawfkeys*, function to format definitions read from function keys, 6-21

*fnextract*, utility function to extract representation of function from package, F-10
*fnname*, utility function to extract function name from canonical representation of function, F-13
*fnpack*, utility function to return packed function from array definition, F-14
*fnprint*, utility function to print function definition on Epson printer, F-8
*)fns*, 3-11
folded-vector field, 13-17, 13-22
*fopen*, utility function to tie a file, creating it if necessary, F-21
forgotten password, 7-2
FORMAT, DOS command, 2-2, 2-4, 2-8
*format*, defined function in WS *fscr*, 13-34
format by example, 3-11
format matrix, FSCR command to read, 13-26
__, FSCR command to set, 13-26
__, FSCR field description, 13-24, 13-25
*format∆nl*, utility function to format names of functions or variables, F-26
*format∆timestamp*, utility function to put □*ts* in standard form, F-26
*format∆ts*, utility function to format timestamps, F-10
formatting, numeric data, 3-11
four-cube problem, E-5
*fprint*, utility function to print description or values of file components, F-22
framing error, ACOM return code, 12-18
free-form, field in FSCR, 13-22
free-form text field, 13-3
French language, APL publications, E-2
*freplace*, utility function for replacing components, with resize as necessary, F-22
frequency, argument to MUSIC AP, 10-17
*from*, utility function for scattered indexing, F-35
*fromto*, utility function to generate consecutive integers in array of ranges, F-35
FSCR, auxiliary processor, full-screen manager, 13-1
__, command summary, 13-35
__, compared with SESM, 13-1
__, compatibility with AP124, 13-1
__, multiple commands, 13-14
__, return code, 13-37
__, return code table, 13-37
__, start parameters, 2-14, 13-40
FSCR.EXE, full screen management AP, 2-9
*fscr*, workspace, 9-16, 13-33
*fscrex*, workspace containing sample full-screen program, 13-3, 13-38
*ftake*, utility function analogous to ↑ for a file, F-22
*ftt*, utility function to format □*rdci* timestamps, F-54
*ftt*, utility function to format □*rdci* timestamps, F-56
full-screen null, 13-18, 13-21
full screen, auxiliary processor, 13-1
__, duration of display, 13-3
full screen edit, FSCR command, 13-29

function, printing definition, 11-12
function call, DOSX, 11-13
function definition, 3-7
function keys, example of definition with □*arbout*, 6-23
  __, read definitions effective under SESM, 6-21
  __, set definitions for use under SESM, 6-21
  __, to release screen during full-screen entry, 13-2, 13-9
  __, to select from menu of options, 13-1
function name, may not be shared, 9-5
function table, produced by ∘. operator, 3-7

G

German language, APL publications, E-1
*gerows*, utility function for determining which rows of a matrix are ≥ , F-36
*get*, function to read session manager's character tables, B-3
*getfields*, defined function in WS *fscr*, 13-35
*getrawfkeys*, function to read SESM definitions of function keys, 6-21
*getscreen*, defined function in WS *fscr*, 13-35
*getsuper*, variant of *get* using DOSX, B-4
Gilman, L., E-1
glyphs, APL character generator, B-6
  __, IBM character generator, B-7
GOOFFER, AP routine to handle received offer, 10-19
GOSHUT, AP shutdown routine, 10-19
GOSVP, request to sign on AP to SVP, 10-17
grab, line from log and copy it to stage, 3-5, 6-10
  __, sequence of lines to be copied to stage, 3-12
grade, function to sort a table, 3-8
*gradedown*, utility function to apply ⍒ to designated axis, F-36
*gradeup*, utility function to apply ⍋ to designated axis, F-36
green, FSCR colour, 13-24
Grenander, U., E-3
growing replace, APL file system, 4-7, 8-10
*gtrows*, utility function for determining which rows of a matrix are > , F-36

H

H (CP8X start parameter), 2-13, 7-2
H (File System start parameter), 2-15, 8-2
half duplex, access control, 9-17
hard disk, building APL system, 2-10, 2-11
hard stop, halt all processing by <Ctrl NumLock>, 6-18
hardware, needed to run SHARP APL/PC, 1-4, 1-5
Hayes Smartmodem, asynchronous communication, 5-1
hertz, tone generated by MUSIC AP, 10-17
hexadecimal, DOS return codes, as subcodes in DOSX return, 11-23
high intensity, value of attribute of FSCR field, 13-24
*highlight*, defined function in WS *fscr*, 13-34
highlighting, attribute of FSCR field, 13-25
  __, FSCR command to re-set, 13-27
hold, file functions to hold shared file, 4-9, 8-5
<Home>, effect during full-screen entry, 13-9

__, effect while editing input stage, 3-5, 6-8
__, move window to show reference line, 3-12, 6-11
__, □*arbin*/□*arbout* code, 6-22
__, summary of actions, A-7
__, within definition of function key, 6-21
horizontal scrolling, 6-12
Δ*rplc*, utility function to replace one string by another, F-6

I

I/O, disable at shutdown of user-written AP, 10-16
I (CP8X start parameter), 2-13
IBM character generator, diagram, A-11, B-7
IBM Personal Computer, SHAPR APL/PC supported for, 0-11, 1-1
IBM 3270 display station, 13-1, 13-8, 13-31
ICS, integrity checksum in ACOM data transmission, 12-6
*identification in use*, 9-7
identifier, file, 4-2, 8-6
idiom library, Finn-APL, E-5
*if*, utility function for conditional execution of an expression , F-27
Ijiri, Y., E-3
illegal overstrike, 6-7
immediate write, FSCR command, 13-28
*immwr*, defined function in WS *fscr*, 13-34
*immwrattr*, defined function in WS *fscr*, 13-35
implicit retraction, shared variable, 9-6
IMS, communication with, 1-3
IN, display in status line when input allowed, 3-7, 6-13
incoming offers, inquiry regarding, 9-8
index, file, report by DOS *size* command, 11-8
index generator, 3-7
indexed selection, 3-8
*indexgen*, utility function to generate multiple sequences of consecutive integers, F-38
*indexofrows*, utility function to look up rows of one matrix in another, F-38
information centre, 1-1
inhibit, reference to shared variable until appropriate reference by partner, 9-11
inhibition, reference to shared variable, depicted in diagram, 9-15
initial selection, to use or set shared variable from user-written AP, 10-11
initial value, variable shared with user-defined AP, 10-5
input interrupt, <Shift ScrollLock>, 6-16
input stage, area of screen where entry prepared, 6-3, 6-4, 6-5
__, □*arbout* code to clear, 6-22
input to full screen display, FSCR command, 13-29
inquire, ACOM command to report status of port, 12-14
<Ins>, key for insert vs. replace, 3-4, 6-8, 6-15, 13-3, 13-8
__, summary of actions, A-9
insertion, within row of a full-screen field, 13-18
insertion vs. replacement of keyed characters, 3-4, 6-8
installation procedure, APL character ROM, 2-6, G-1
instruction pointer, set by *call* to execute machine-language program, 11-16

INT INTPOST, to schedule normal task, user-written AP, 10-2
integer, APL function ⍳, 3-7
integrity checksum, ACOM data transmission, 12-6
  __, algorithm, 12-9
intensity, attribute in FSCR field description, 13-24
  __, FSCR command to re-set, 13-27
*intensity*, defined function in WS *fscr*, 13-34
intercharacter timeout, in ACOM transmit/receive command, 12-7
interface, multiple, to DOSX, 11-1
  __, shared variable, 9-1, 9-17, 13-10
interlock, shared variable, 13-12
interlock delay, 9-16, 9-17
interrupt, hardware, in user-written APs, 10-2
  __, incoming data via ACOM, 12-11
  __, signal from keyboard, 6-16, 6-18
interrupt handling, MUSIC control of timer, 10-17
interrupt task, task dispatcher, 10-2
interruption, setting shared variable instead of reading it, 9-15
INTPOST, software interrupt, with user-written AP, 10-2
invalid sequence, SVP return code to user-written AP, 10-14
invisible, input to SESM, 6-19
  __, value of intensity of FSCR field, 13-24
invisible columns, until window moved left or right, 6-12
Iverson, K.E., E-1, E-2

J

Janko, W.H., E-2
Japanese language, APL publication, E-2, E-3

K

key names, 6-7
key numbers, PC keyboard, 6-6
keyable, effect of skip attribute in FSCR, 13-25
keyable field, attribute in FSCR field description, 13-24
keyboard, blank diagram, B-9
  __, diagram, 6-6, A-4
  __, during full-screen display, 13-7
  __, flexible support, 1-4
  __, session manager's tables, B-1
  __, SHARP APL/PC, 6-6, A-2
  __, union, C-1
)*keyboard*, difference between PC and 370, D-6
keyboard input, interpretation, 6-2
keyboard input buffer, type ahead during full-screen mode, 13-10
keyboard input to full screen display, FSCR command, 13-29
keyboard state, changing, 6-15
keynumbers, diagram, B-8
<Keypad-minus>, grab line and replace prior contents of stage, 3-5, 6-10
<Keypad-plus>, effect during fill-screen entry, 13-9
  __, keyname, A-3
  __, summary of actions, A-9
<Keypad-plus> and <Keypad-minus>, contrast manner of copying pegged line, 3-9

<Keypad0->, keyname, A-3
__, summary of actions, A-8
<Keypad plus>, grab line from log, insert into stage, 6-10
<Keypad 5>, summary of actions, A-8
keys, indicated by angle brackets, 0-12
__, names used, 0-2, A-2
keystroke, session manager's table to map keystrokes to actions and characters, B-2
keystroke table, B-11

L

L (ACOM start parameter), 2-13
languages other than English, adjustment of session manager character tables, B-1
*lastones*, utility function to preserve only the last of each group of 1s in a vector, F-38
*lastzeroes*, utility function to preserve only the last of each group of 0s in a vector, F-39
<Left>, cursor movement, 6-9
__, keyname, A-3
__, summary of actions, A-8
__, window movement, 6-12
*leftjustify*, utility function to rotate leading dummy values from left edge, F-39
Legrand, B., E-2
length, name, difference between PC and 370, D-5
__, of each word in a text, 3-8
__, variable shared with DOSX, 11-2
*length error*, 8-10
length indicator incorrect, SVP return code to user-written AP, 10-13
library, difference between PC and 370, D-2
__, file, function to list, 4-5
library number, means to identify drive on which workspace stored, 7-1
light pen, not supported by FSCR, 13-1
limit, file size, 4-8
line, contrasted with row, 6-1
__, unit of storage in session log, 6-1
line drop, ACOM return code, 12-17
__, report when ACOM start times out, 12-4
LINK, DOS, to create executable object code from output of DOS assembler, 11-16
list, synonym for "vector", 3-6
*listfns*, utility function to format function definitions, F-8
listing, sample assembler program, 11-20
LK, in status line indicates scroll lock, 3-13, 6-14
*)load*, 3-11
__, effect on shared variable, 9-6
local variable, sharing, 9-6
lock, scrolling of windows, 6-15
locked workspace, temporary circumvention, 2-16
log, move to end or beginning, 3-12
__, record of prior entries and results, 3-3, 6-1, 13-2
low-priority task, task dispatcher, 10-2

lower case, APL alphabet, 1-4
lower window, 6-2, 6-3, 6-11, 6-13
  __, vertical scrolling, 6-11
LPB--, reserved name for DOS file intended for printer, 11-11
LPT1 (pseudofile), 11-12

M

M (FSCR start parameter), 2-15, 13-40
machine-language program, called via DOSX, 11-16
main keyboard, diagram, A-4
mainstore address, SVP operations by user-written AP, 10-7
majuscule, APL alphabet, 1-4
majuscule letters, C-2
make directory, DOSX function call, 11-14
*makeΔnl*, utility function to form matrix from vector delimited by blanks, F-27
*makeΔstdΔfilename*, utility function to generate filename in standard form used by □*lib* etc., F-28
mark, in ACOM start command, 12-4
*maskparts*, utility function function for partitioned masking , F-40
mathematical experiments, text in APL, E-3
matrix, data for FSCR field transmitted as ravel, 13-18
*matrix*, utility function to assure matrix, F-14
matrix product, 3-7
maximum table, 3-7
McDonnell, E.E., E-5
*memberrows*, utility function to report whether each row of one array occurs in another, F-40
memory, amount needed to run SHARP APL/PC, 1-4
  __, direct read and write by *peek* and *poke* commands, 11-15
memory allocation, user-written AP, 10-15
memory map, environment of machine-language program executed under DOSX, 11-17
menu selection, full screen, 13-1
*mesh*, utility function to mesh catenated arrays, F-41
messages, FSCR errors, function to decode, 13-38
Microsoft Corporation, Disk Operating System User's Guide, E-5
minimum separation of fields, 13-1
minus, keyname, A-2
minuscule, APL alphabet, 1-4
minuscule letters, C-2
MKDIR, DOS command, 2-11
  __, DOS command compared with DOSX *mkdir*, 11-14
*mkdir*, DOSX command to make a directory, 11-14
*mod*, function to modify data in tables used by session manager, B-3
modem signals, 12-20
modified fields, reported in result of FSCR command 3, 13-30
modularity, benefit of shared variables, 1-3
monochrome display, with IBM PC, 1-5
movement of cursor, 6-8, 6-9
)*msg*, difference between PC and 370, D-6
multiple lines, copying from log to stage, 3-9
multiple tasks, clone ID to distinguish, 9-7
  __, task dispatcher, 10-2

Musgrave, G.L., E-3
MUSIC, initialization of AP, 10-17
MUSIC.ASM, sample auxiliary processor assembler source file, 10-1, 10-17
MUSIC.EXE, sample auxiliary processor object code, 10-17
MYTICKINT, interrupt task for MUSIC AP, 10-18

N

N-task, difference between PC and 370, D-3
name, difference between PC and 370, D-5
__, file, distinguishing APL from DOS, 4-2
__, restrictions imposed by SVP, 9-5
__, shared with DOSX, 11-1
__, shared with FSCR, 13-13
__, shared with user-written AP, 10-6
*name∆sort*, utility function to sort names in telephone-book order, F-8
names, for keys, A-2
__, list, in sample program, 13-4
__, offered to shared with AP, 9-5
*nations*, file containing sample data, 3-8
__, sample APL read via DOSX, 3-10
negative sign, 3-6
newline, delimiter of lines in session log, 6-1
__, required in some print operations, 11-12
newline vs. cr, in translations between ASCII and APL, 11-6
newsletter, 1-5
no offer, SVP return code to user-written AP, 10-13
no share, SVP return code to user-written AP, 10-14
*no space*, message, 7-2
no value, SVP return code to user-written AP, 10-14
non-APL programs, sharing to communicate with, 9-2
normal, intensity of FSCR field, 13-24
normal end, SVP return code to user-written AP, 10-13
normal task, task dispatcher, 10-2
noskip, parameter in FSCR field description, 13-25, 13-27
__, value of FSCR field description, 13-22
not available, return code to user-written AP, 10-14
not signed on, SVP return code to user-written AP, 10-14
*notmemberrows*, utility function to report whether each row of an array occurs nowhere in another, F-41
NP, status indicator when numeric pad enabled, 6-7, 6-13
null, distinction between two members of □*av*, 13-21
__, to replace deleted positions in a full-screen field, 13-18
null vs. blank, row-trailing positions in full-screen field, 13-19
nulls, midline, difference in effects in PC and IBM 3270, 13-23
__, trailing, 13-21, 13-25, 13-27
<Num Lock>, 13-7
number, file component, effect of dropping preceding components, 4-7
number in use, SVP return code to user-written AP , 10-13
number of tied files, 8-9
numbers, sample program for full-screen display, 13-4
__, used to tie files, 8-5
numeric field, attribute in FSCR field description, 13-24

numeric keypad, 3-14, 6-6
numeric lock, 6-7
<NumLock>, 6-7, 6-8, 6-12, 6-15
__, summary of actions, A-7

O

O-backspace-U-backspace-T, input interrupt <Shift ScrollLock>, 6-16
odd parity, in ACOM start command, 12-4
*)off*, effect on tied files, 4-5, 2-10, 8-1
__, effect on user-written APs, 10-16
offer, to share variable with AP, 9-4
offer not found, SVP return code to user-written AP, 10-13
offer to share, by user-written AP, 10-9
one-row field, mechanism to obtain free-form FSCR field, 13-22
online bulletin board, I.P. Sharp Associates, 1-5
open, native file, 3-10
*openfx*, utility function to open printer pseudofile, F-14
openro, DOS command to open file for read only, 11-6
openrw, DOS command to open file to read and write, 11-6
operating system, auxiliary processor for communication with, 1-3, 3-9, 11-1
order, arrange list in, 3-9
order of overstrikes, 6-7
orderly communication, access controls to assure, 9-15
Orth, D.L., E-3
Osada, J., E-2
over-writing, in full-screen field, 13-18
overbar, <Keypad minus> while using numeric keypad, 6-8
overflow, ACOM buffer, 12-17
overrun, ACOM return code, 12-18
overstrike, Alt backspace to form overstruck character , 3-2
overstriking, 6-7
ovesrtrike, session manager's table of characters thus formed, B-2

P

P (CP8X start parameter), 2-13
P (SVP start parameter), 2-16
<PA>, correspondence between PC keys and 3270 keys, 13-10
*packshow*, utility function to display contents of package neatly, F-55
*packshow*, utility function to display contents of package, F-56
Pakin, S., E-1
*pandreduce*, utility function for partitioned ∧/, F-42
*pandscan*, utility function for partitioned ∧\, F-42
*para*, utility function to format vector , F-6
parameters, start, 2-13
__, SVP commands, from user-written AP, 10-8
parity, in ACOM start command, 12-3, 12-4
parity error, ACOM return code, 12-18
passnumber, access to file, 4-10, 4-11
passnumer, file, 4-2
password, workspace, 7-1
__, workspace, temporary circumvention, 2-16

path, optional parameter in DOS file command, 11-5
PCCMD, field of AP's per-command control vector, 10-5
PCFLAGS, field of AP's per-command control vector, 10-5
*pcnews*, via SHARP timeshared APL, 1-5
PCP1, PCP2, PCP3, fields of PCCMD containing arguments and results for SVP call from user-written AP , 10-5, 10-6
PCVID, field of AP's per-processor control vector, 10-4
PCVNID, field of AP's per-processor control vector, 10-4
PCVPOST, field of AP's per-processor control vector, 10-2, 10-4, 10-17
PCVTBP, field of AP's per-processor control vector, 10-4
PCVTCS, field of AP's per-processor control vector, 10-4
PCVTDS, field of AP's per-processor control vector, 10-4
PCVTSS, field of AP's per-processor control vector, 10-4
*peek*, DOSX command to read machine memory, 11-15
peg, location reported in status line, 6-13
  __, mark line of log as reference line, 6-10
  __, mark line of log for copying to stage, 3-3, 6-10
  __, moved or made visible by <PgDn> and <PgUp> keys, 3-4, 6-10
*pegscan*, utility function for partitioned =\, F-43
per-processor shutdown of APs, 1, 10-18
PERCMD, at each call, to SVP from user-written AP, 10-5, 10-7
  __, type of structure (control block) in user written auxiliary processor, 10-4
PERCMD PCP2, in result of SVPON function, with user-written AP, 10-3
period, <Ins> key while using numeric keypad, 6-8
  __, keyname, A-2
permission codes, file access matrix, 4-11, 8-12
PERPCV, AP's per-processor control vector, 10-4, 10-17
  __, task under task dispatcher, with user-written AP, 10-2
  __, type of structure (control block) in user-written AP, 10-4
PERPROC failure, SVP return code to user-written AP, 10-14
PERPROC Index, unique identifier for task, with user-written AP , 10-2
PERPSHUT, per-processor shut-down of APs, 10-16, 10-18, 10-19
PERSCV, AP's per-share control vector, 10-5, 10-6
  __, type of structure (control block) in user-written auxiliary processor, 10-4
PERSHARE failure, SVP return code to user-written AP, 10-14
personal computer, connection to another, 12-1
Personal Computer AT, SHARP APL/PC supported for, 0-11, 1-1
Personal Computer AT/370, SHARP APL/PC supported for, 0-11
Personal Computer XT, SHARP APL/PC supported for, 0-11, 1-1
Personal Computer XT/370, SHARP APL/PC supported for, 0-11
<PF>, correspondence between PC keys and 3270 keys, 13-10
  __, indication whether pressed in result of FSCR command 3, 13-30
<PgDn>, move peg, 3-5, 6-10
  __, move peg, 3-5, 6-10
  __, summary of actions, A-9
*pgradedown*, utility function for partitioned ⍦, F-43
*pgradeup*, utility function for partitioned ⍋, F-44
<PgUp>, summary of actions, A-8
physical storage of files, 8-10
pink, FSCR colour, 13-24

*pkgmatch*, utility function to extend ≡ to packages, F-8
PLAY, normal task used by MUSIC AP, 10-17
*plengths*, utility function to report length of each partion of a boolean vector, F-44
*plescan*, utility function for partitioned ≤\, F-44
*pltscan*, utility function for partitioned <\, F-45
plus, keypad, during full-screen entry, 13-9
*plusscan*, utility function for partitioned +\, F-49
*pmaxreduce*, utility function for partitioned ⌈/, F-45
*pmaxscan*, utility function for partitoned ⌈\, F-46
*pminreduce*, utility function for partitioned ⌊/, F-46
*pminscan*, utility function for partitioned ⌊\, F-47
*pnescan*, utility function for partitioned ≠\, F-47
point of view, shared variable state and control, 9-8
*poke*, DOSX command to write to machine memory, 11-15
Polivka, R.P., E-1
population density, example of computation, 3-9
*porreduce*, utility function for partitioned ∨/, F-48
*porscan*, utility function for partitioned ∨\, F-48
port, ACOM command to inquiry about status , 12-14
__, difference between PC and 370, D-5
__, in ACOM transmit/receive command, 12-6, 12-8, 12-9
__, specified in ACOM start command, 12-3
Portable Personal Computer, SHARP APL/PC supported for, 0-11, 1-1
)*ports*, difference between PC and 370, D-6
position, field, 13-16
post, task to task dispatcher, with user-written AP, 10-2
post bits, indicate why task posted, with user-written AP, 10-2
posting, task, under PC task dispatcher, 10-2
power table, 3-7
PPI, PERPRC Index, user-written AP, 10-2, 10-3, 10-9
*pplusreduce*, utility function for partitioned +/, F-49
precision, formatted numeric data, 3-11
*preverse*, utility function for partitioned ⌽, F-50
primary alphabet, 1-4
print, DOS command, 3-11
print screen, <Shift PrtSc>, 6-16
printer, translate option in DOSX file operations, 11-5
__, translation to display APL listings, 11-11
printer support, 11-11
*printfx*, utility function to print character string on Epson printer, F-14
priority of tasks, task dispatcher, 10-2
PRN, reserved name for DOS file intended for printer, 11-11
*prn*, name of DOS file to be printed, 3-10
processor ID, auxiliary processor, 10-4
product table, 3-7
program, machine-language, 11-16
program symbol set, (unused) attribute of FSCR field, 13-25
programming services, I.P. Sharp Associates, 1-5
progressive product, 3-7
*proindexof*, utility function for progressive index, F-50
protected field, attribute in FSCR field description, 13-24
<PrtSc>, summary of actions, A-6
PSS (program symbol set), 13-24

Q

Q (ACOM start parameter), 2-13
Q (DOSX start parameter), 2-14, 11-2
Q (File System start parameter), 2-15, 8-2
Q (FSCR start parameter), 2-15, 13-40
Q (SVP start parameter), 2-16
quad-prime input/output, effect on full screen display, 13-3
⎕*ai*, difference between PC and 370, D-5
⎕*append*, write new component to file, 8-4
⎕*appendr*, write new component to file, 8-4
⎕*arbin* (system function), B-16
⎕*arbin*, return codes, 6-18
⎕*arbin*/⎕*arbout*, difference between PC and 370, D-5
 __, difference in behaviour, 6-17
 __, integer or character argument, 6-17
 __, to communicate with session manager, 6-17
 __, to define function keys, 6-22
⎕*arbout*, B-16
⎕*av*, C-1, C-2
⎕*avail*, function to test whether file system is running, 4-9
 __, report availability of APL file system, 8-4
⎕*create*, create new APL file, 8-4
⎕*drop*, remove components from end of file, 4-7, 8-4, 8-10, D-6
⎕*erase*, erase an APL file, 8-4
⎕*fhold*, function to hold shared file, 4-9, 8-5
⎕*fmt*, 1-4
⎕*hold*, difference between PC and 370, D-3
 __, function to hold shared file, 4-9, 8-5
⎕*ht*, difference between PC and 370, D-4
⎕*lib*, function to list file library, 3-8, 4-5, 8-5
⎕*names*, function to report names of tied files, 4-6, 8-5
⎕*nums*, function to report numbers used to tie files, 4-6, 8-5
⎕*pw*, 3-14, 6-1
⎕*rdac*, read access matrix of APL file, 8-5
⎕*rdci*, function to read control information of APL file component, 8-6
⎕*read*, function to return value of file component, 3-8, 3-9, 8-6
⎕*rename*, change name of an APL file, 4-8
 __, difference between PC and 370, D-3
⎕*replace*, function to replace value in a component of a file by another value, 4-7, 8-6
⎕*resize*, function to revise file's size limit, 4-8, 8-6
⎕*run*, difference between PC and 370, D-5
⎕*signal* (system function), 1-4
⎕*size*, function to report size of an APL file, 4-8, 8-7
⎕*sp*, difference between PC and 370, D-4
⎕*stac*, function to set file access matrix, 8-7
⎕*stie*, function for shared use of a file, 4-9, 8-7
⎕*svc*, control of reference to shared variable, 9-4, 9-8
 __, interlock variable shared with FSCR, 13-12
 __, interlock variable shared with MUSIC, 10-19
⎕*svn*, function to set or inquire about clone ID, 9-7
⎕*svo*, degree of coupling reported, 9-8
 __, handling offer received by MUSIC AP, 10-19
 __, offer to share with FSCR, 13-12

__, offer to share with ACOM, 12-1
__, parallel wtih SVP command SVPSVO, 10-8
__, shared variable offer to establish link with auxiliary processor, 3-9, 9-4
□*svq*, shared variable query, 9-8
□*svr*, function to withdraw share offer, 3-11, 9-6, 10-19
__, parallel with SVP command SVPSVR, 10-8, 10-19
□*svs*, function to report shared variable state, 9-5, 9-8
__, interpretation of possible values, 9-9
□*tie*, function for exclusive use of a file, 4-9, 8-7
□*trap* (system variable), 1-4
□*untie*, function to dissolve link between APL file and tie number, 3-9, 8-7
□*ws*, difference between PC and 370, D-5
query, to mean "any character" in DOS file request, 11-5
quote, keyname, A-2
quote marks, to indicate character data, 3-8

R

R, display in status line while work in progress, 3-7, 6-9, 6-13
Ramsey, J.B., E-3
*randomize*, utility function to set □*rl*, F-5
*rank error*, 8-10
rate, communication, in ACOM start command, 12-3
read, components of an APL file, 4-5, 8-6
__, DOS file command, 11-7
__, portion of DOS file, 11-4, 11-7
read/write, DOS file, 3-10
read a file, 3-8
read and write, DOS command to open file, 11-6
read entire screen, FSCR command, 13-32
read fields, FSCR commands 5, 30, 31, 13-31
read format matrix, FSCR command, 13-26
read modified fields, FSCR command, 13-31
read only, DOS file, 3-10, 11-6
*readdata*, defined function in WS *fscr*, 13-35
*readdatax*, defined function in WS *fscr*, 13-35
*readformat*, defined function in WS *fscr*, 13-34
*readscreen*, defined function in WS *fscr*, 13-35
ready to send, effect of ACOM stop command, 12-13
receive count, in ACOM transmit/receive command, 12-7
receive count expired, ACOM return code, 12-18
receive special character, ACOM return code, 12-18
__, in ACOM transmit/receive command, 12-7
receive timeout expired, ACOM return code, 12-18
reclaim disk space, utility function, 8-10
record, variable length assumed in APL file system, 4-7
record size, underlying storage of APL files, 8-10
*recurrence*, utility function for progressive growth with multitplier and addend at each step, F-51
red, FSCR colour, 13-24
redundant reference to shared variable, 9-15
reference, set, use, distinction between terms, 9-4
reference diskette, list of files, 2-7
reference diskettes, preparing, 2-2

reference line, fixed position in log, 3-12, 6-10
__, move window to show, 6-11
*reformat*, defined function in WS *fscr*, 13-34
registers, initial values set by *call* DOSX function, 11-16
registration, SHARP APL/PC, 1-5
release, key to signal that typed entry is ready for execution, 3-1, 3-6, 6-9, 13-2
release of shared variable, consequence of successful use or set by user-written AP, 10-12
__, following initial selection but no use or set by AP, 10-12
Release 17, of SHARP APL, 0-11, 1-1
remove directory, DOSX function call, 11-14
*removetrail*, utility function for removing trailing columns or elements from an array, F-51
*removetrail*, utility function to remove trailing elements or columns, F-51
rename, APL file, 4-8, 8-6
__, difference between PC and 370, D-3
__, DOS file, 11-9
replace, keyed character on screen, 6-8, 6-15, 13-3, 13-8
__, value in a component of a file by another value, 4-7, 8-6
replace mode, 6-8, 6-15, 13-3, 13-8
reservation, file, 4-3, 4-8
__, file, function to change, 4-8
reset, session manager, by <Esc>, 6-1, 6-14
__, system, by <Ctrl Alt Del>, 6-16
resize, file, 4-8, 8-6
result, returned by DOSX *call* of machine-language program, 11-18
RET FAR, instruction to return from normal task, with user-written AP , 10-2
retraction, by AP to match retraction from APL, 9-6, 10-19
__, shared variable, 9-6
__, shared variable, with user-defined AP, 10-5, 10-10
return, from machine-language program called under DOSX, 11-16
<Return>, keyname, A-2
__, Possible result reported by FSCR command 3, 13-30
__, □*arbin*/□*arbout* code, 6-22
__, release signal, 3-1
__, summary of actions, A-6
__, to release screen (under FSCR) , 13-1
__, to release stage (under SESM), 6-9
__, within definition of function key, 6-21
return codes, ACOM, 5-3, 12-17, 12-18, 12-19
__, from □*arbin*, 6-18
__, from DOSX, table, 11-23
__, from FSCR, summary, 13-37
__, from SVP to user-written AP, 10-13
__, shared variable interface to FSCR, 13-14, 13-35
__, shared variable interface to DOSX, 11-1
reversal, APL function, 3-6
reverse, caps/lower case, 6-15
reverse video, value of FSCR "highlighting" attribute, 13-25
reversing half-duplex, access control, 9-17
__, shared variable control between APL and MUSIC, 10-19
<Right>, cursor movement, 6-9

__, keyname, A-3
__, summary of actions, A-8
__, window movement, 6-12
*rightjustify*, utility function to right-justify each row of an array, F-52
*rightjustify*, utility function to rotate trailing dummy values from right edge, F-52
ROM, APL characters, 1-5, 2-6, A-11
__, APL diagram, B-6
__, characters supplied by IBM, B-7
__, installation procedure, 2-5, G-1
__, vulnerable to static electricity, G-2
ROM table, mapping from □*av* to character generator glyphs, B-1
Rose, Allen J., E-1
row, contrasted with line, 6-1
__, of a field displayed by FSCR, 13-18
__, of screen in display of log, 6-1
row-structure, field displayed by FSCR, 13-17
RS232, asynchronous communications protocol, 12-1
RS232 ports, 1-3, 5-1, 12-1
__, 103 501 1201
RTS, turned off by ACOM stop command, 12-13
*runlengths*, utility function, F-52
*runlengths*, utility function to return length of each sequence of unchanged values, F-52
Russian language, APL publications, E-1

S

S-task, difference between PC and 370, D-3
S (SVP start parameter), 2-16
SAF, name extension for APL file, 2-9
*sample*, listing of full-screen function, 13-4
sample program, full screen, 13-3
)*save*, 3-11
__, effect of shared variable, 9-19
SAW, name extension for APL workspace, 2-9
SC, status line indicates lower window not at last line of log, 6-14
scalar, applied to all elements of an array, 3-6
scanning for offers, user-written AP, 10-9
screen, display generated by FSCR, 13-2
__, internal storage used by FSCR , 13-6
__, shape and size, 13-16
screen areas, 6-2, 6-3
screen buffer, FSCR, 13-6
scroll lock, 3-13
<Scroll Lock>, 13-7
scrolling, lock/unlock, 6-14
__, move segment of log displayed, 3-3, 6-11
__, use of Down, Shift down, Up, Shift up keys, 3-12
<ScrollLock>, lock or unlock vertical scrolling of lower window, 6-11
__, summary of actions, A-7
SCVALUE, field of AP's per-share control vector, 10-5
SCVCHRON, SVP's record of share chronology, 10-6, 10-9
SCVCOUNT, field of AP's per-share control vector, 10-5
SCVNAME, name of variable offered to user-written AP, 10-9

SCVNAMEL, length of name offered to user-written AP, 10-9
SCVNID, field of AP's per-share control vector, 10-5
SCVNOTHR, clone ID of partner of user-written AP, 10-6, 10-9
SCVOTHER, SVP's record of user-written AP's partner's ID, 10-6, 10-9
SCVPO, per-offer counter, SVP, 10-6, 10-9
SCVSOS, field of AP's per-share control vector, 10-5
sealed workspace, 7-2
*secret*, function to make input invisible, 6-19
segment/offset address, contrasted with mainstore address, 10-7
*select*, DOSX command to report or select default drive, 11-13
selection by indexing, 3-8
semicolon, keyname, A-2
  __, separates DOSX commands and return codes from data, 11-2
sending and receiving, shared variable as means, 9-4
*separate*, defined function, WS *fscrex*, 13-38
SESM, default state of screen and keyboard, 6-14
  __, session manager, 1-4
  __, session manager, compared with full screen manager, 13-1
*sesm*, workspace to modify session manager's character tables, B-2
session log, 1-2, 6-1, 6-2
  __, set size, 2-16
  __, show last line, 6-11
  __, show oldest line, 6-11
Session Manager, 1-2, 6-1
  __, actions, B-11
session manager, 6-1
  __, keyboard and character tables, B-1
  __, reset state, 6-14
session variables, difference between PC and 370, D-4
set, use, reference, distinction between terms, 9-4, 10-11
*set*, function to set new session manager character tables, B-3
set cursor , FSCR command, 13-28
set data transfer, SVP command, 10-11
set format matrix, FSCR command, 13-26
set initial selection, SVP command, 10-11
set of shared variable, requires AP to make intial selection before using, 10-11
set shared variable, 13-5
  __, interlock controls, 9-9
set translate table, DOSX command, 11-10
*setcursor*, defined function in WS *fscr*, 13-35
*setfkeys*, function to associate text with function keys, 6-20
*setrawfkeys*, function to set SESM definitions of function keys, 6-21
*setsuper*, variant of *set* using DOSX, B-4
*settrtab*, DSOX command to set new translate table, 11-10
SFS.EXE, SHARP APL/PC file system, 2-9
SFS start parameter, maximum number of tied files
shadowing, shared variable, 9-6
shape, field, 13-16
  __, field, displayed by FSCR, 13-24
share tie, function for shared use of a file, 4-9, 8-8
*shareap*, defined function in ws *fscr*, 13-34
*sharecheck*, sample program to verify that variable is shared , 9-8
shared file, 4-9

shared variable, effect of )*save* or )*continue*, 9-19
__, interface to ACOM, 12-1
__, interface to DOSX, 11-1
__, set and use as communication, 13-5
__, state change, 9-18
__, who has set?, 9-9
__, who is aware of current value?, 9-9
shared variable control, inquiring about, 9-8
__, point of view, 9-8
shared variable delay, 9-16, 9-17
shared variable inquiry, regarding offers from others, 9-8
__, user-written AP, 10-5
shared variable offer, auxiliary processor, 10-5
Shared Variable Processor, 1-3, 9-1
__, outline of how it works, 9-3
__, start parameters, 2-16
shared variable state, inquiring about, 9-8
__, point of view, 9-8
shared variable storage failure, 10-13
shared variables, 9-1, 12-1
sharing, to link with ACOM processor, 5-2
shift, reversed by Alt Ctrl keys, 3-4
<Shift>, 3-1
__, 6-4
shift case, 6-4
<Shift Del>, delete first of line, 6-8
__, □*arbin*/□*arbout* code, 6-22
__, within definition of function key, 6-21
<Shift Down>, 3-12
<Shift Up>, 3-12
sign-on, SVP from user-written AP, 10-8
sign on to timeshared system, 5-1
*simple*, utility function to determine whether an array is simple, F-28
simplex, access control, 9-17
single-variable interface, FSCR, 13-13
sixteen-bit architecture, 1-1
size, DOS file command, 11-8
__, file, effect of replacing file component by another requring more space, 4-7
__, file, encoded in control information for each component, 4-9, 8-6
__, file, function to report, 4-8, 8-7
*sizes*, utility function to report formatted table of names and sizes of objects, F-29
skip, attribute of FSCR field, 13-25
__, FSCR command to re-set, 13-27
slash, keyname, A-2
sleep, until state change, 9-18
Smith, Adrian, E-1
*smpl*, defined function, WS *fscrex*, 13-38
sorted table, 3-8
*sortlocal*, utility function to sort local variables in function header, F-6
space, available to session log, 6-1
__, parity specification in ACOM start command, 12-4

__, reclaimed by utility function, 8-10
*space*, utility function to calculate bytes required to execute , F-5
Spadavecchia, V., E-5
Spanish language, APL publications, E-1
speaker, to sound musical tones, 10-17
Spence, R., E-3
*squeeze*, utility function to remove all leading, trailing, and redundant elements from vector, F-53
*squeeze*, defined function to reclaim waste space in a file, 8-10
__, utility function to remove repeated, leading or trailing dummy elements, F-53
squish-quad, display of unrecognized character, 13-21
stage, area of screen where entry prepared, 6-2, 6-3, 6-5, 13-2
starmap, publication, E-5
start, ACOM command to ready port for communication, 12-3
start APL, 2-10, 2-11
start parameters, ACOM, 2-13
__, CP8X, 2-13
__, DOSX, 2-14, 11-2
__, DOSX, reported by function *version*, 11-13
__, File System (SFS), 2-15, 8-2
__, for APL system and its processors, 2-13
__, FSCR, 2-14, 13-40
__, Shared Variable Processor (SVP), 2-16
*startfscr*, defined function, WS *fscrex*, 13-38
startup of user-written AP, terminate but stay resident, 10-15
state, keyboard, 6-15
__, shared variable, 9-5, 9-10
state change signal, shared variable, 9-17
state of the keyboard, status-line display , 3-4
static electricity, hazard during installation of ROM, G-2
static window, 3-13, 6-3
statistics, text in APL, E-3
status, display during full-screen mode, 13-10
status line, absent during full screen, 13-2
__, display by session manager, 6-2, 6-3, 6-13
__, display of keyboard state, 3-4
__, during full-screen mode, 13-10
__, moved by <Alt PgUp> and <Alt PgDn>, 6-14
stickers, keytop, for APL characters, 6-6, 6-7, A-2
stop, ACOM command, 12-13
stop address, using DEBUG and CALLTEST, 11-19
stop bits, in ACOM start command, 12-4
stop cpu, <Ctrl NumLock>, 6-16
storage, APL files, 8-10
__, available to session log, 6-1
__, deallocation following shut-down of an AP, 10-16
__, file, effect of replacing file component by another requring more space, 4-7
storage deficiency, SVP, 10-13
storage protection exception, SVP return code to user-written AP, 10-14
storage space, effect of dropping file components, 8-4
stored representation of screen, 13-6
*stringfind*, utility function to find all occurrences of one vector in another, F-54

*stringfind*, utility function to find beginning of each occurrence of one vector in another, F-54
strong interrupt, Ctrl Break Break, 3-10, 6-18
  __, to escape from shared-variable wait, 9-16
structure, control block, in user-written auxiliary processor, 10-4
STSC Inc., APL character generator, 2-6
  __, APL users meeting, E-4
subcodes, in return from DOSX, 11-23
*subtotal*, utility function to insert first-axis subtotals into an array, F-29
subtraction sign, 3-6
subtraction table, 3-7
summary, FSCR commands, 13-36
summation, 3-7
SUPER, parameter for size of session log, 6-1
  __, start parameters, 2-15
suppress display, □*arbin*/□*arbout* command, 6-19
surrogate name, 9-5, 13-10
*sv interrupt*, 3-10, 9-16
SVCNAME, name of variable shared by user-written AP, 10-6
SVID, field in AP's per-share control vector, 10-5
SVP, 9-1, 10-1
  __, command syntax, from user-written AP, 10-8
  __, operation invoked from user-written AP, 10-7
  __, required to use full screen manager, 13-5
  __, start parameters, 2-14, 2-16
SVP.EXE, shared variable processor, 2-9
SVPCDT, SVP command to copy data transfer, 10-11
SVPNDT, voluntary release of control of shared variable, 10-12
SVPON, sign-on to SVP from user-written AP, 10-8
SVPON function, user-written AP, 10-3
SVPSDT, SVP command to set data transfer, 10-11
SVPSVC, inquiry regarding access control by user-written AP, 10-11
  __, setting access control by user-written AP, 10-11
SVPSVO, AP's per-share control vector, 10-5
  __, inquiry regarding degree of coupling, 10-10, 10-19
  __, offer to share by user-written AP, 10-9
SVPSVQ, AP's per-share control vector, 10-5
  __, scanning for offers to user-written AP, 10-9
SVPSVR, shared variable retract by user-written AP, 10-10
SVPUDT, SVP command to use data transfer, 10-11, 10-19
SVPUIS, use initial selection, user-written AP, 10-11, 10-19
SVZ--, prefix of SVP return codes, 10-8
*swap error*, message, 7-2
System/370 personality, on PC XT/370 and AT/370, 0-11
system commands, examples, 3-11
system reset, <Ctrl Alt Del>, 6-16

T

T, SFS start parameter: number of tied files, 2-14
T (File System start parameter), 2-16, 8-2
<Tab>, cursor movement while editing input stage, 6-9
  __, effect during full-screen entry, 13-9
  __, keyname, A-2

__, summary of actions, A-6
<Tab> , 3-5, 13-3
table, indexing by, 3-8
__, produced by ∘. operator, 3-7
task dispatcher, deallocation of storage used by APs, 10-16
__, support of concurrent tasks, 10-2
task ID, difference between PC and 370, D-5
task posting, user-written AP, 10-2
tasks, multiple concurrent, and shared files, 4-9
terminal, use of PC as, 5-1
)*terminal*, difference between PC and 370, D-6
terminate but stay resident, interrupt required in starting user-written AP, 10-15, 10-17
termination, user-written AP, 10-16
TEST.--, names of files containing test program, data, and result used by CALLTEST, 11-19
text, free-form, 13-3
text processing, simple example, 3-8, 3-9
third alphabet, 1-4
Thorstensen, J., E-4
tie, dissolved by □*untie*, 4-4
__, number associated with file while in use, 4-3, 8-7
tie-numbers, APL files, 8-5
*tied*, utility function to report tied files and their tie numbers, F-30
tied files, functions to report names and tie numbers, 4-6
__, maximum number, 8-9
timeout, ACOM return code, 12-18
__, first character, in ACOM transmit/receive command, 12-7
__, intercharacter, in ACOM transmit/receive command, 12-7
times table, 3-7
timestamp, encoded in control information for file component, 4-9, 8-6
toggle, reversing control of keyboard state, 3-4
tools, to install ROM, G-1
trailing, nulls vs. blanks, FSCR command to re-set field attribute, 13-27
trailing blanks, FSCR command to re-set treatment, 13-27
__, treatment options during full-screen editing, 13-19, 13-20, 13-21, 13-25
transfer software, 5-10
transition diagram, possible shared-variable states, 9-14
translate, DOSX command, 11-9
translation, between DOS and APL data encodings, 3-10, 11-5, 11-8
__, DOSX command to establish new table, 11-10
transmission, final character, in ACOM transmit/receive command, 12-7
transmission count expired, ACOM return code, 12-17
transmit/receive, ACOM, 12-6
transmit break, ACOM command, 12-11
trouble report, sent via shared variable from FSCR, 13-13, 13-14
*ttf*, utility function to form □*rdci*[3]-format-encoded timestamp vector from matrix of □*ts*-formatted timestamps, F-55
*ttf*, utility function to encode date and time in form used by □*rdci*, F-56
two-variable interface, auxiliary processor, 13-14, 13-15

__, FSCR, 13-14
type, attribute of a field displayed by FSCR, 13-24
__, FSCR command to re-set, 13-27
typeahead, 3-7, 6-9
__, delete contents of buffer, 6-9
__, full-screen mode, 13-10

U

U (ACOM start parameter), 2-13
U (CP8X start parameter), 2-13
U (DOSX start parameter), 2-14, 11-2
U (FSCR start parameter), 2-15, 13-40
Uchiyama, A., E-2
underscore, value of FSCR "highlighting" attribute, 13-25
union keyboard, 1-3, 6-6, C-1
universal access, zero in file access matrix, 4-10
*unless*, utility function for conditional execution, F-30
unrecognized character, display, 13-21
untie, file, 4-4, 4-7, 8-7
<Up>, cursor movement, 3-5, 6-9
__, keyname, A-2
__, movement of window, 3-5, 6-11
__, summary of actions, A-7
upload files, PC to mainframe, 5-1
upper window, 3-13, 6-3, 6-11, 6-13
use, contrasted with set of shared variable, 10-11
use, set, reference, distinction among terms, 9-4
use data transfer, SVP command, 10-11
use initial selection, SVP command, 10-11
use shared variable, 13-5
__, interlock controls, 9-9
__, requires AP to make initial selection before using, 10-11
user-written machine-language program, called from DOSX, 11-16
user number, implicit in offer to share, 9-7
__, of user who wrote file component, 8-6
__, set as start parameter, 2-15
Users Meeting, E-4
*utility*, workspace of APL utility functions, 4-9, F-1
utility functions, 11-11

V

validation, data shared between APL and user-written AP, 10-6
*value error*, 3-3, 10-6
variable-length record, in APL file system, 4-7
variable too large, SVP return code to user-written AP, 10-14
*)vars*, 3-11
vector, 3-6
__, folded, to produce free-form display in FSCR, 13-22
*version*, DOSX command to report DOS version, 11-13
vertical scrolling, 6-11
VTAM, communication with, 1-3
*vtom*, utility function returns matrix with one string per row, F-54
*vtom*, utility function to return matrix from vector with delimiters, F-55

W

W (CP8X start parameter), 2-13
W (DOSX start parameter), 2-14, 11-2
wait, until state change, 9-18
wait for input, FSCR command, 13-29
warranty, SHARP APL/PC, 1-5
width, parallel printer interface via DOSX, 11-2
window, current, reported in status line, 6-13
  __, dynamic vs. static, 3-12, 6-2, 13-2
  __, select current, 6-15
withdraw share offer, 3-11
words, cut operator to separate text, 3-8
workspace, and drive on which available, 7-1
  __, automatically loaded at startup, 2-16
  __, compared with file for storage, 4-1
  __, diskette to store, 2-8
  __, identifiers, 7-1
  __, size when saved or loaded, 7-2, D-3
workspaces, 7-1
  __, saved, 1-2
wrap, enable/disable, 6-15
  __, state of windows, reported in status line, 6-13
wrapped vector field, 13-22
wrapping of log display, <Alt NumLock>, 6-12
write, buffered, FSCR command, 13-28
  __, DOS file command, 11-7
  __, immediate, FSCR command, 13-28
  __, native file, 3-10, 11-4
*write*, defined function in WS *fscr*, 13-34
write/wait/read, FSCR command, 13-29
*writeattr*, defined function in WS *fscr*, 13-34
*ws full*, 8-10
  __, from attempt to )*save* or )*continue* with shared variable, 9-19
*ws locked*, temporary circumvention, 2-16

X

XON/XOFF, flow control, 12-4

Z

zero, file access matrix, 4-10

Numeric

1, FSCR command to set field formats, 13-26, 13-33
2, FSCR command to transmit new field data to buffers, 13-27, 13-33
3, FSCR command to write screen, wait for input, and report, 13-29
4, FSCR command to write field data and display all fields, 13-27, 13-33
5, FSCR command to read specified fields, 13-30, 13-34
6, FSCR command to re-set field type, 13-27, 13-33
7, FSCR command to re-set field intensity, 13-27, 13-33

8-bit data, transparent translation in ACOM transmit/receive command, 12-6
9, FSCR command to read format matrix, 13-26, 13-33
11, FSCR command to sound alarm, 13-28, 13-34
12, FSCR command to set cursor position, 13-28, 13-34
16, FSCR command to re-set skip and trailing blank attributes, 13-27, 13-33
16-bit architecture, IBM PC, 1-1
24, FSCR command to re-set field colour, 13-27, 13-33
26, FSCR command to re-set field highlighting, 13-27, 13-33
27H, interrupt to terminate but stay resident, 10-15, 10-17
39, DOS function call, 11-14
30, DOS function call, 11-13
__, FSCR command to read modfified fields as matrix, 13-31, 13-34
31, FSCR command to read modified fields as enclosed arrays, 13-31, 13-34
35, FSCR command to re-set attributes of individual characters, 13-29, 13-33
37, FSCR to write character attributes to buffers, 13-29, 13-33
40, FSCR command to read entire screen, 13-32, 13-34
700, processor ID of ACOM, 12-1
1002, weak interrupt, 6-16
1003, strong interrupt, 6-16
1004, input interrupt, 6-16
3000, default user number, 2-15
3270, different treatment of insertions and trailing nulls, 13-20, 13-32
__, different treatment of deletions within a folded-vector field, 13-22, 13-32
__, display station, 13-1, 13-31
__, restriction on adjacent fields, 13-17, 13-32

Printed in Canada
February 1985
Publication Code: 0227 8502 E2